Scripting Modeling Languages

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

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

ABSTRACT

Domain-Independent Models are mainly used for documentation purposes and are most of the time too complex 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 propose to develop a mechanism that allows the creation of modeling languages that will be directly executed into software applications. We inspire ourselves from dynamic languages, especially scripting languages, and adapt 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 claim 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

1. INTRODUCTION

One of the main advantages of using 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 [9].

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 unexperienced 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 [10], a new screen helper for World of Warcraft, or even a new campaign for a game if the game's architecture allows it [6]. 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 (*cf.* figure 1).

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 for even more unexperienced developers. The idea is to create scripting modeling languages that are easier to learn and easier to modify, yet with the same flexibility as dynamic languages [5]. Scripting modeling languages are intended to be mapped to scripting languages in real-time. This could open new perspectives and indirectly spread the use of dynamic languages to a sphere of less-experienced developers. Furthermore, 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. This software development methodology, known as *model-driven software engineering* (MDE) [12], addresses the intrinsic complexity of software-intensive systems by raising the level of abstraction, and by hiding the accidental complexity of the underlying technology as much as possible [2].

Our approach could be interesting in the scope of MDE. UML is one of the most popular domain-independent software modeling languages [4]. 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 to maintain consistency between the different views during evolution [13, 15]. 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 aim to integrate executable models as small parts of the software, as opposed to executing a model representing the entire software. In the traditional approach, a model represents the global application architecture and design at a high level, and the programmer needs to refine the model with a textual programming language. In our approach, the model does not represent the whole design of the application but only the behavior of one of its subsets. The main idea is to delegate specific tasks to an external model the way we do with scripting languages.

These days, more and more software applications use external scripting languages to extend their features. Section 2 briefly summarizes the concept and the advantages of using these scripting languages. Section 3 describes our vision of a scripting modeling language and its technical aspects. Section 4 highlights, with the help of two examples, the main advantages of using a model that controls the flow of an application.

2. SCRIPTING LANGUAGES

Scripting languages are often embedded in applications to extend their features. They are usually adapted for large applications where there is a need for modularity and where it could be 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 [9]; (4) scripting languages usually have the same capabilities as independent 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 the application. One can make some changes in the script and see the impact on the application behavior in real-time.

Scripting languages are an easy way to involve the customer in the software development process. This allows 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 [1].

There are many success stories involving scripting 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[1].

Some well-known commercial proprietary software is also using scripts. It 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. SCRIPTING MODELING LANGUAGES

Some studies have shown that using visual models is advantageous for a better understanding and a faster modification of software [16, 7]. However, it is unclear how hard it is to create a software design model that could define every aspect of a software application.

[1]http://wiki.blender.org/index.php/Extensions:2.4/Py/Scripts

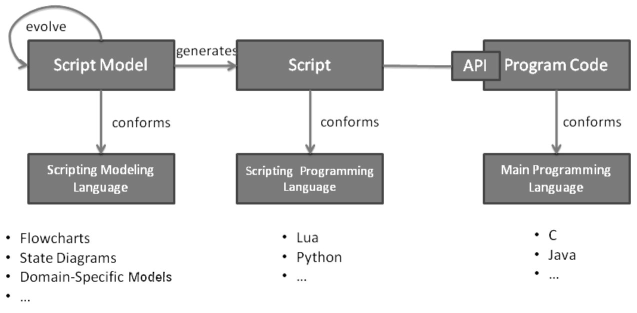

Figure 1: The technical solution we use to execute scripting modeling languages

We propose to merge the advantages of both modeling and scripting languages. We have seen that scripting languages improve the software modularity. We claim that it is useful and possible to keep the modularity properties of scripting 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 [11]. In our solution, the model is considered as a primary artifact of the software and is directly executed.

Technical solution

We want to have the possibility to use and combine different scripting modeling languages. They will be transformed *on-the-fly* to a textual scripting language (like Lua). There will be domain-independent modeling languages, like flowcharts, and domain-specific modeling languages. Core 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 scripting modeling process in figure 1. The left part of figure 1 represents the script model that conforms to the scripting modeling language. This modeling language can take various forms like flowcharts, state diagrams, or domain-specific models. Each time the script model evolves, the new related script, which conforms to a scripting language (Lua, Python, ...), is generated. This script is used by the main program through the dedicated API. This API is created using the main programming language.

Like a textual script, a script model only has access to methods, attributes, or even classes, that are specified by

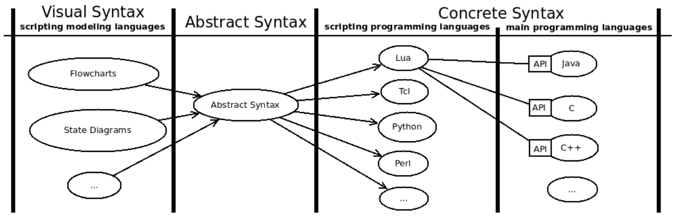

Figure 2: The abstract syntax is used to make the bridge between scripting modeling languages and scripting programming languages

the API. For example, if we need a script model to sort a table, the API will be containing attributes like the `table` itself and methods like `length(table)` and `swap(table, i, j)`. If the API is correctly defined, it will not be able to call other functions that are not useful for sorting.

In practice, we operate in two steps. The first step is the generation of a textual script from a script model. It has to be flexible enough to handle a wide range of modeling languages (including new ones that are domain-specific) and any scripting language. To avoid having to create a mapping from all modeling languages to all scripting languages, we use an abstract syntax to make the bridge (*cf.* figure 2). This abstract syntax is closer to the textual scripts than the scripting modeling language. To transform a script model into the abstract syntax and to transform the abstract syntax into a script, we use the ASF+SDF Meta Environment [14]. ASF+SDF Meta Environment is a syntax/semantic analysis and transformation tool that is able to transform abstract syntax trees (ASTs) to other ASTs.

The second step consists of creating a library adapted to a main programming language. This library must allow developers to create dedicated APIs for script models. This library is also responsible to load script models and to transform them into textual scripts in a transparent way. The library needs to be created for each main programming language we may want to use. It will be built on the existing libraries for scripting languages such as LuaJava [3], which can load a Lua script in Java.

4. EXAMPLES

We illustrate the advantage of our approach with two concrete examples. They concern a *mobile museum guide* application [8]. 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 unexperienced 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 directly taken into account. A second advantage of scripts in this context is that the museum manager can use them to create a visit scheduling adapted to the visitors' 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 (flowchart).

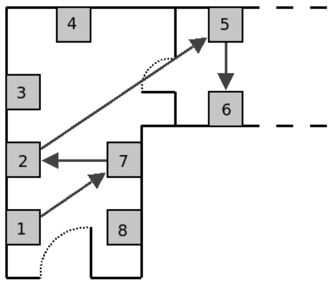

Figure 3: The museum plan with a superimposed visit schedule using our scripting modeling language

```
if artwork1.position == visitor.position and next == artwork1 then
        presentation(artwork1)
        next = artwork7
elseif artwork2.position == visitor.position and next == artwork2 then
        presentation(artwork2)
        next = artwork5
else if artwork3.position == ...
...
else
        print("go to artwork" .. next.id)
end
```

Figure 4: The generated Lua code for the script model of figure 3

For better readability and space considerations, the two examples developed in this section are very simple but they can get more complex depending on the user's needs.

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 more 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 each visit when the museum is restructured, a scripting modeling language can be used, just like the one shown in figure 3. The scripting model incorporates the museum plan and allows the manager to modify the schedule of the visit by adding or removing arrows.

Once the script model of our schedule is finished, it is loaded in the main language by our library, and the corresponding textual script is generated and executed on-the-fly. Using Lua as scripting language, the generated code is displayed on figure 4. Note that the code is 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 automatically.

Interactive artwork presentation

In the mobile museum guide, each artwork needs a presentation that may contain text, image, video, and sound, that must be defined. 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. Besides

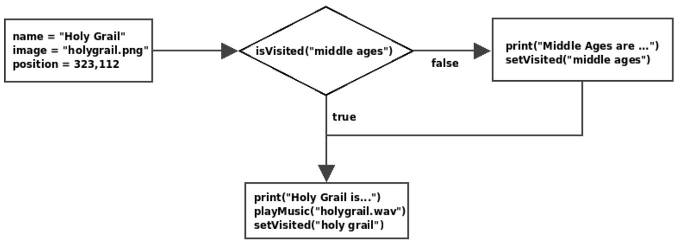

Figure 5: The (short) presentation of an artwork using our scripting modeling language

```
artwork.name = "Holy Grail"
artwork.image = "holygrail.png"
artwork.position = 323, 112

if not isVisited("middle ages") then
        print("Middle Ages are...")
        setVisited("middle ages")
end

print("Holy Grail is...")
playMusic("holygrail.wav")
setVisited("holy grail")
```

Figure 6: The generated code for the script model of figure 5

that, when a visitor stands in front of a medieval artwork, a quick introduction about the Middle Ages must be triggered. However, this introduction must be triggered just once. In order to avoid needing to hard-code this kind of behavior and to allow the manager to change it in an simple way, a flowchart like the one in figure 5 will be used.

Once this flowchart is finished, it is loaded in the main language by our library each time a visitor is close to an artwork. The textual script is generated and executed on-the-fly, and the artwork presentation is given to the visitor. Using Lua as scripting language, the generated code is displayed in figure 6.

5. CONCLUSION

We have shown in this article that it is possible to raise the level of abstraction of *textual* scripting languages to obtain scripting *modeling* languages. The main advantage of this is that less-experienced developers can easily change an application's behavior or add a behavior without installing a complete tool-chain or even recompiling the application.

Our 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.

Through two concrete examples concerning a *mobile museum guide*, we have seen that, in the context of concrete problems, raising the level of abstraction can improve the modularity of the application. This modularity improvement enables museum managers or even historians to modify the application's behavior themselves just by modifying the script models and without bringing any change to the core of the application. This flexibility and the use of script models will make software evolution an easier and cheaper task.

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)*

6. REFERENCES

[1] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004.

[2] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* 20th anniversary edition, 1995.

[3] C. Cassino, R. Ierusalimschy, and N. Rodriguez. LuaJava-a scripting tool for Java. *Arxiv preprint cs/9903018*, 1999.

[4] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] G. Kappel, J. Vitek, O. Nierstrasz, S. Gibbs, B. Junod, M. Stadelmann, and D. Tsichritzis. An object-based visual scripting environment. *Object-Oriented Development, ed. D. Tsichritzis, Université de Genève*, 1989.

[6] G. Lyrio and R. Seixas. Using Lua as Script Language in Games Coded in Java. In *Proceedings of The North American Simulation and AI in Games Conference - GAMEON-NA, EUROSIS, Montreal, Canada*, 2008.

[7] R. Navarro-Prieto. Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human Computer Studies*, 54(6):799–830, 2001.

[8] R. Oppermann and M. Specht. Adaptive support for a mobile museum guide. In *Proceedings of the Conference on Interactive Applications of Mobile Computing.* Citeseer, 1998.

[9] J. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE computer*, 31(3):23–30, 1998.

[10] A. Peck. *Beginning GIMP*, chapter Plug-ins and Scripting, pages 435–480. Springer, 2009.

[11] B. Rumpe. Executable Modeling with UML. A vision or a Nightmare. *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle. Idea Group Publishing, Hershey, London*, pages 697–701, 2002.

[12] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, pages 25 – 31, February 2006.

[13] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World scientific, 2001.

[14] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, et al. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Compiler Construction*, pages 365–370. Springer.

[15] R. Van Der Straeten, T. Mens, J. Simmonds, and

V. Jonckers. Using description logic to maintain consistency between UML models. In *UML 2003*, page 326. Springer Verlag, 2003.

[16] K. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109–142, 1997.