ContextLua

Dynamic Behavioral Variations in Computer Games

Benjamin Hosain Wasty[1] Amir Semmo[1]
Malte Appeltauer[2] Bastian Steinert[2] Robert Hirschfeld[2]
Software Architecture Group
Hasso-Plattner-Institut, University of Potsdam, Germany
[1]{first.last}@student.hpi.uni-potsdam.de
[2]{first.last}@hpi.uni-potsdam.de

ABSTRACT

Behavioral variations are central to modern computer games as they are making the gameplay a more interesting user experience. However, these variations significantly add to the implementation complexity. We discuss the domain of computer games with respect to dynamic behavioral variations and argue that context-oriented programming is of special interest for this domain. This motivates our extension to the dynamic scripting language Lua, which is frequently used in the development of computer games. Our newly provided programming constructs allow game developers to use layers for defining and activating variations of the basic gameplay.

1. INTRODUCTION

Context information is central to dynamic variations of behavior in running applications. This information mainly depends on a large range of dynamic attributes, whether defined by changes in the system's environment, the specific actions of an actor, or the current state of the system itself.

Context-oriented programming (COP) is a promising programming paradigm to permit and improve the maintainability, robustness and reusability of systems that must fit in these highly dynamic environments [9]. In particular, COP supports the expression of dynamic variations at language level that might be conceptually orthogonal to each other. Consequently, based on the principle that specific variations in the execution of an application can be encapsulated in layers, COP is of special interest for program domains that comprise much variety and imply context-dependent behavioral variations.

One domain that highly depends on behavioral variations are computer games. While the basic gameplay mechanisms often are comparatively straight-forward to manage, additional features and behavioral variations can enrich the gaming experience. These behavioral variations originate from the broad range of actors and modules (Figure 1). They often influence several modules of the system (Section 2) and require remarkable effort to keep the implementation complexity manageable. COP provides adequate support for defining and activating behavioral variations, but the applicability of COP for the domain of computer games has not yet been discussed.

In this paper, we discuss the need for better support of behavioral variations in the domain of computer games and describe how COP concepts are able to facilitate, amongst others, the implementation of artificial intelligence (Section 2). We further present a new implementation of COP support for Lua, a C++ based scripting language that is frequently used for programming high-level logic in computer games (Section 3). In particular, we build our implementation on top of Lua's meta-mechanisms for facilitating the handling of behavioral variations at runtime. Specifically, metatables and metamethods are used as a basis for supporting a simple and straight-forward implementation of object-oriented and context-oriented programming concepts.

2. MOTIVATION

Most of today's computer games represent complex architectures that build on the premise of allowing players to freely roam inside a virtual world. As a result of these open world scenarios, completely linear gameplay is to a large extent only of minor interest. Instead, variations in gameplay and behavior are taken into consideration in order to remove easily predictable behavior by adapting the gameplay to specific changes in the environment and actions of the player, which is commonly related to artificial intelligence [7, 13].

However, a broad range of actors and components can be identified in game worlds (Figure 1), whose orthogonal behavioral variations inevitably lead to highly complex models of artificial intelligence:

- *Environmental* entities are mostly referred to as components that shape the virtual world. We classify them into static entities like terrain, buildings or vegetation objects and real-world phenomenons, e.g. specific weather conditions.
- *Players* are the main source for variations in a game, whereas monsters comprise complex autonomous systems to react to the specific modes and actions of a player.
- *Physics* play a major role for adding realism to a game. They especially impact and constrain the actions made by characters.

ECOOP '2010 Maribor, Slovenia EU

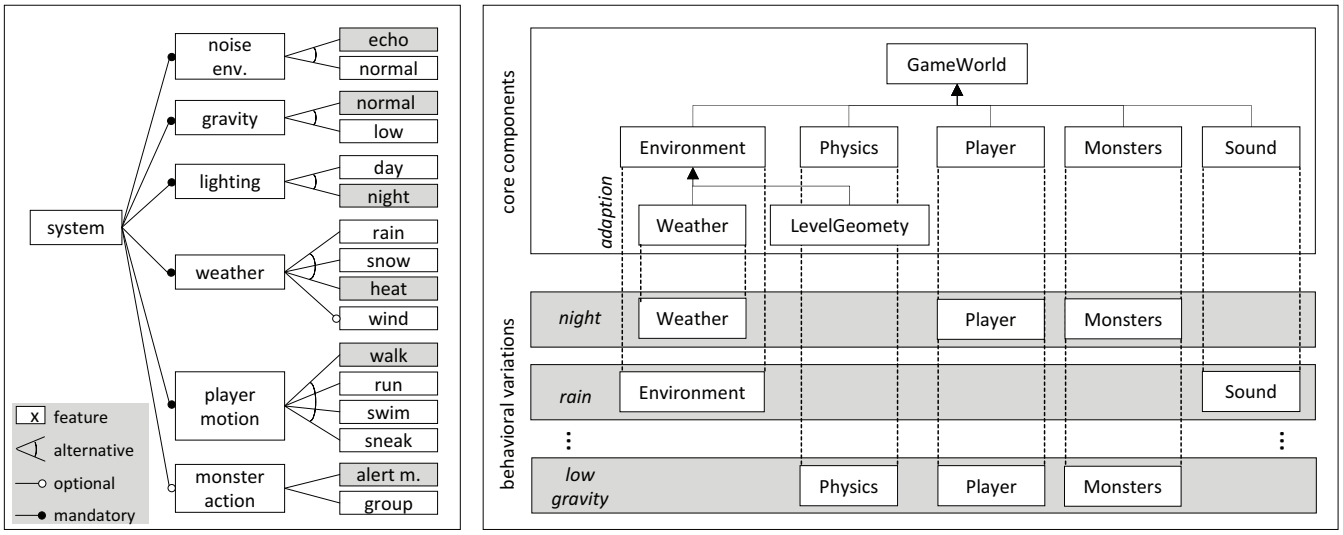


Figure 1: Overview of the features and components of a game world. *left:* feature variations. *right:* components and selected feature variations (grey).

- *Sound* effects add depth to a game and mostly reflect or enhance the current settings of the environment and situation the player is in.

Most importantly, behavioral variations that originate from specific entities influence the behavior of other entities, e.g. low gravity affects the movement of characters, the night mode affects the visibility range and distinctive weather modes affect the sound setting. A more detailed example is described below.

2.1 Example

A popular way for modeling artificial intelligence in computer games is using finite state machines (FSM). Within these models, knowledge is modeled by states, whereas actions are constrained by rules and modeled by transitions. Figure 2(a) depicts a self-contained state transition diagram that highlights the specific actions of a monster in a game world.

When we look at the actions of a monster, 4 states are appropriate to model basic behavior. The activation of these states mainly reflect changes in the environment, and dependent on which state the monster is in, it reacts to its current situation:

1. *Idle State*: A monster initially resides in the idle state. In this state it only performs a standard animation like looking around and turning its head. The decisive factor for leaving the idle state and attacking is when a player comes close to it. This behavior is triggered when the distance is smaller than a predefined sensing radius.

2. *Attack State*: While attacking, a monster constantly checks its health points. If the health points are low and it has no health potion, it flees. When it has no health points left, the monster becomes unconscious. Furthermore, it might return to the idle state as soon as the player is out of range (for example when he has fled).

3. *Flee State*: When the monster flees, it runs away until it is out of danger, i.e. the distance to the player is large enough to recover. The monster then goes back to the idle state.

4. *Unconscious State*: When a monster is unconscious, a counter is started that counts down a predefined period of time. Then the monster awakes and returns into the idle state.

Although these states and transitions describe some intelligent reactions of a monster to changes in the environment, e.g. attacking when a minimum range is maintained or fleeing when wounded, the transitions are rather straightforward and predictable for the player. Consequently, variations are required on top of this basic behavior that originate from other variations in the game world.

2.2 Variation

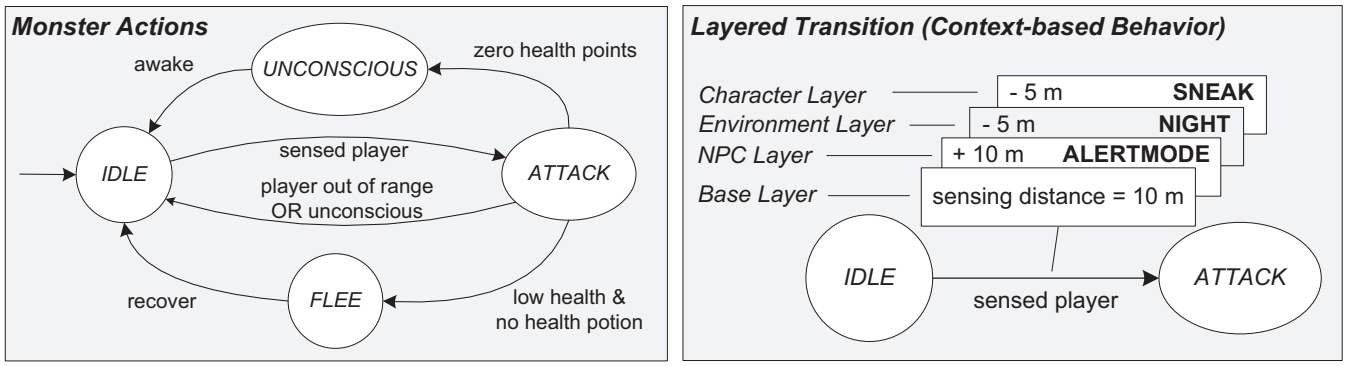
Orthogonal variations that result from other components in the game world most likely influence decisions of monsters in certain situations. A detailed example is depicted in Figure 2(b). Here the decision to attack a player is influenced by multiple variations. As we take a sensing distance of 10 meters as a trigger for going into the attack state, this basic behavior might be influenced by:

- An alert mode, where monsters are alarmed and especially conscious about approaching players. In this case, the sensing distance is increased by 10 meters.

- A night mode, where the visibility range is decreased by 5 meters.

- When the player is in sneak mode, i.e. the noise is kept at a lower level and the player moves more slowly. In this case, the player is harder to be spotted and the sensing distance decreases by 5 meters.

Besides that example, further behavioral variations can be implied. For example, when several monsters are close



(a) (b)

Figure 2: State transition diagram with actions of a monster and behavioral variations.

together they could go into a group mode where they work together and attack collectively. In this case, the transition of the attack state to the flee state might be omitted, as long as it is assumed that monsters in a group never flee from a player. A further variation addresses the flight behavior itself - at night a shorter distance would be required to recover and go into idle state. Furthermore, one might think of a complete new behavior where hiding in shadowed areas might be sufficient for recovering. Further variations generally have influence on rendering, e.g. movement of characters varies (sneak mode), lighting adapts to certain weather conditions (rain) or the sound settings adapt to the environment (howls of wolves at night).

2.3 Motivation

The behavioral variations presented in the last section can be modeled as layers, as they are independent from each other and each one enhances or redefines the basic behavior of a monster (Figure 2(b)). We argue for modeling behavioral variations as layers, since without them each variation in the state transition diagram would have to be modeled explicitly. In general, if a transition is influenced by n factors, a total of

$$x = \begin{cases} \sum_{k=0}^{n} \frac{n!}{(n-k)!}, & \text{regarding order of factors} \\ \sum_{k=0}^{n} \frac{n!}{k!(n-k)!}, & \text{else} \end{cases}$$

transitions would be required to fetch all possible combinations. For example, 8 transitions would be required from idle to attack state, as of commutativity. Furthermore, a new state would be required in the example of the varied flight behavior at night. Layers however, have the opportunity to completely redefine a basic behavior in a state.

Context-oriented programming (COP) facilitates the concept of layers at language level. It is based on the assumption that specific variations in the execution of an application are encapsulated in layers, which can be referred to as first-class entities. Dependend on the current context, those layers can be enabled or disabled during runtime in order to aggregate context-dependent behavioral variations. Distinctive variations can then be active simultaneously in different scopes allowing the system to respond to those contexts in parallel.

3. CONTEXTLUA

Lua was created in 1993 by Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo [11]. Their main motivation was to create a language that on the one hand permits lightweight and embeddable scripting, whose syntax is easy to understand for non-programmers, and on the other hand permits high portability on diversive platforms [12]. Along with high compatibility to C/C++, Lua is frequently used in computer graphics applications to separate the graphics engine from interactive functionality (e.g. user interface and game logic) and provide an appropriate level of reusability of the core functionalities.

In the following section we describe how object-oriented and context-oriented concepts can be implemented on the basis of Lua's meta concepts. We especially highlight a simple and straight-forward implementation that is based on Lua's metatables and metamethods for handling behavioral variations at runtime.

3.1 Metatables and Metamethods

Tables are the only data structure mechanism Lua offers and are used for the representation of ordinary arrays, sets, symbol tables and also packages. Moreover, they can be used to implement higher-level concepts through use of their meta-features. They are implemented as associative arrays, i.e. each data type can be assigned as a key (except for *nil*).

The central meta concept is the *metatable*. A metatable is a regular table with some special fields: The *metamethods*. Any table can be associated as a metatable to any other table (*setmetatable() / getmetatable()*). Each metatable can be shared across multiple tables, e.g. for sharing specific behavior.

Metamethods are predefined operators. Among these are methods that describe arithmetical (e.g. *__add* for "+") or relational functions (e.g. *__eq* for "==") that are called implicitly when an operator is used. Additionally, there are metamethods that allow controlling table access:

1. *__index* is called when a field of a table is accessed that does not exist,
2. *__newindex* is called when as a value is assigned to a table field whose old value was *nil*,
3. *__call* is executed when a table is called like a function.

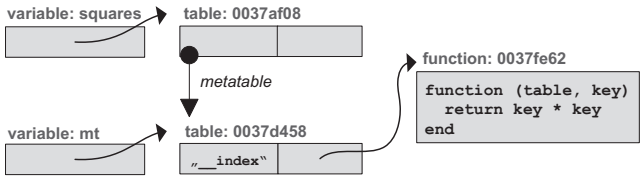


Figure 3: Example where *__index* is used to return the squared value of the accessed key (field).

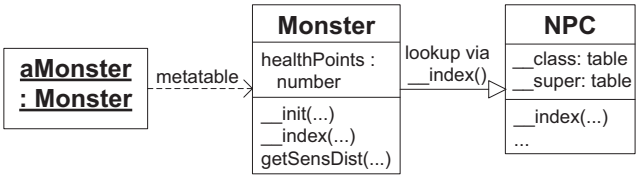


Figure 4: Relation between instances, classes and superclasses (*NPC*: Non-Player Character).

Figure 3 shows an example that uses *__index* to return the squared value of the requested key at each access (e.g. *squares*[7] yields 49)[1]. Furthermore, *__index* and *__newindex* can reference a table instead of a function. Thus, a hierarchical lookup mechanism for the implementation of object-oriented concepts with inheritance can be achieved in a cheap and simple way.

3.2 Object-oriented Programming

In the following, we define an object-oriented concept based on tables and metamethods, where objects are modeled as regular tables and metamethods facilitate a simple lookup mechanism for inheritance. Three characteristics contribute to the design of objects:

1. *Identity*: Independent from its current state, each table has its own identity,

2. *State*: A state of an object is represented by key-value pairs of a table,

3. *Operations*: In Lua functions are first-class values. Consequently, functions can be stored as regular values in a table and operate on its current state.

Listing 1 shows a function *createClass* that supports single inheritance and Figure 4 shows a concrete example. When *createClass* is called, the class table is initially created and references to the class and its superclass are set (Lines 2–4), e.g., for allowing introspection functionality. A class has a predefined function *new* that has an arbitrary number of arguments as its signature (Line 5). This function creates new instances by creating new tables (Line 6) and automatically calls an (optionally) predefined constructor *__init*, passing on the arguments given to *new* (Line 8).

For the lookup of class and instance members, we use Lua's metatables and metamethods. In our concept, every instance has its class assigned as metatable (Line 6). We then define the metamethod *__index* as a simple lookup forwarding to the class itself (Line 12). When a member of an instance is to be accessed, the request will be forwarded to the class if it is not found in the instance. All methods are stored prototypically in the class table. Instance variables are defined in the constructor. Furthermore, each class is assigned a metatable, whose metamethod *__index* points to the superclass (Line 13), i.e. the lookup continues in the superclass when members can not be found in a class.

As a conclusion, the presented class concept features a simple and straight-forward implementation for inheritance.

[1]Graphic is based on `http://phrogz.net/lua/LearningLua_ValuesAndMetatables.html`.

```
1  function createClass(superClass)
2     local self = {}
3     self.__class = self
4     self.__super = superClass
5     function self:new (...)
6        local instance = setmetatable({}, self)
7        if self.__init then
8           self.__init(instance, ...)
9        end
10       return instance
11    end
12    self.__index = function(t, k) return self[k] end
13    local mt = {__index = self.__super}
14    return setmetatable(self, mt)
15 end
```

Listing 1: Implementation of a class concept with single inheritance. A class is metatable of an object table and a superclass metatable of a class table.

Furthermore, it supports creating new instances and defining a constructor that is called automatically at object creation.

Static methods can also be defined - the *self* parameter just needs to point to a class instead to an instance when calling the function. Additionally, multiple inheritance can be supported by allowing to specify an arbitrary number of superclasses and redefining the lookup of *__index* so that it iterates over the superclasses when a member is not found in the current class.

3.3 Context-oriented Programming

Based on the described OOP features, we present additional constructs for COP: the ability to define layers for the encapsulation of specific variations and for dynamically activating these layers during runtime.

3.3.1 Usage of ContextLua

First, we show the usage of context-oriented features in association with the example in Section 2.2.

Layer Definition.

The behavioral variations described by layers are defined by partial method definitions:

```
1  function Monster:getSensingDistance()
2     return 10
3  end
4
5  function Monster:Night_getSensingDistance()
6     return proceed() - 5
7  end
```

At the beginning, we define the base method *getSensingDistance()*. Afterwards a variation of this method,

Night_getSensingDistance(), is defined in the layer *Night*. The method name consists of the layer name, an underscore for separation and the name of the base method.

The layered method in the above example uses the *proceed* method. This method calls the next appropriate method in the current layer composition. When only the *Night* layer is active, this would be the base method. *Proceed* can receive an arbitrary number of arguments, which are then passed on. How proceed is used defines how the current layer treats the base functionality respectively other layers that have been activated before. As a conclusion, behavior can be replaced, extended or modified.

Layer Activation.

Layers can be activated by using the *with* method:

```
1  with(Night, function() -- or: with("Night",...
2     print(monster:getSensingDistance())
3  end)
4
5  with({Night, Sneak}, function()
6     print(monster:getSensingDistance())
7  end)
```

The first parameter is the layer or a table of layers to be activated. The second parameter is a function which encapsulates the code that should be executed with these layers being active. The layer identifiers are global variables which point to a string that contains the layer name. So it is also possible to pass a string containing the layer name.

Analogous to the *with* method, the *without* method can be used for deactivating layers. There also are two methods allowing to activate layers without specifying a scope: *activateLayer* and *deactivateLayer*. These are also used internally to implement *with* and *without*.

3.3.2 Implementation

For implementing the creation of new layers (i.e. partial methods), we make use of the *__newindex* metamethod (see Section 3.1). Such a method is added to the metatable of each class. It is always called when a member is added to a class and is responsible for properly registering the member with the class, so that it can be quickly retrieved later during method lookup (based on current layer and method name). Furthermore, when a new layer is encountered, a global variable, whose identifier is the layer name, is defined. The currently active layers are managed by a global table called *ActiveLayers*, which is used as a simple list and is manipulated by the *with* and *without* methods.

In other COP implementations like JCop, the list of active layers is saved in a thread-local variable, as is described in [2]. This is done in order to support the activation of different sets of active layers in different threads, so that they do no interfere with each other. Lua supports no true, concurrent multithreading; the thread type actually represents a coroutine, which only enables collaborative multithreading. That is why we decided for a simple global variable. The layer-aware method lookup is implemented in the *__index* metamethod, which is a modified version of the implementation in Section 3.2.

For the implementation of *proceed*, we use the Lua debug library. With it, the whole call stack can be accessed and manipulated. We retrieve the function that called *proceed* and its *self* argument, i.e. the object on which the function was called. We then determine the name and layer of the calling function and the next function to be called, based on the current layer composition. This function is then called with *self* and the parameters passed to *proceed*.

```
1  -{ block:
2     local function builder (block)
3        local layers, expr = unpack(block)
4        return +{
5           with(-{layers}, function() -{expr} end)
6        }
7     end
8
9     mlp.lexer:add{"with"}
10    mlp.expr:add {
11       "with", mlp.expr, "do", mlp.block, "end",
12       builder = builder
13    }
14 }
```

Listing 2: Syntax extension with Metalua. Definition of a *with* statement(9-13) that is transformed into a call to the *with* function (2-7).

3.3.3 Metalua - Syntax Extension

Our interface for using COP features consists of a code convention for method definition and the helper functions *with* and *without*. However, an explicit syntax for creating, activating and deactivating layers would be preferable. It should look as follows:

```
1  layer Night do
2     function Monster:getSensingDistance()
3        ...
4     end
5  end
6  ...
7  with Night do
8     print(monster:getSensingDistance())
9  end
```

A way to implement such extensions is using the Lua extension *Metalua* [6]. It consists of a compiler written in Lua and a set of predefined language extensions. The compiler generates bytecode compatible with the standard Lua virtual machine. A central feature of Metalua is the dynamically extensible parser, which we used to implement the aforementioned *with*-statement in a few lines of code (Listing 2).

The reason for defining the slightly less elegant interface with pure Lua first instead of using Metalua from the start is, that we did not want to have Metalua as a prerequisite for using ContextLua. This is mainly because Metalua still is in alpha stage (though quite stable) and currently not actively maintained.

3.3.4 Discussion

With our prototypical implementation we showed that COP concepts can be implemented elegantly in Lua with relatively little code. In computer games, performance is of critical importance. To bring our implementation beyond the prototype stage, the performance should be benchmarked and the code optimized if necessary. It may be advantageous to implement critical code parts such as method lookup in C/C++ for optimal performance. This would not affect the code structure significantly, as Lua is designed to allow easy interaction with C/C++ code..

4. RELATED WORK

In the following, we discuss COP implementations related to ContextLua and alternative specifications of state-based behavior.

4.1 COP Language Implementations

In previous work [3], we describe two main implementation strategies for layer-based recomposition that take place either at *composition time*, where method references are relinked at any occurrence of *with* and *without*, or at *execution time*, where method versions are selected just before their execution. Our ContextLua implementation follows the latter strategy by extending the metatable-based lookup mechanism. This very late binding allows for alternative layer lookup algorithms, such as event-based compositon [2].

Several other COP extensions also implement recomposition at execution time. The Lisp extension *ContextL* [4] is based on the Common Lisp Object System; *ContextPy* [10] and *PyContext* [14] use Python's reflective API. Their implementation is similar to ContextLua as they also use the meta programming capabilities of their host language. Other extensions, such as the Java-based language *JCop* [2] cannot be implemented in such a straightforward way, since their base language does not provide direct access to its lookup mechanism as Lua does.

4.2 State-based Behavior

As described in Section 2.1, a popular way for modeling artificial intelligence in computer games is using finite state machines (FSM). Straight-forward implementations of FSMs use conditional clauses, i.e. *switch* or *if* statements. More sophisticated approaches use object-oriented programming concepts in order to benefit from dynamic binding and reducing complexity on implementation level (state pattern [8]). Specific approaches even support states at language level (UnrealScript [5]), where arbitrary functions can be redefined for particular states.

Unfortunately, these solutions generally do not scale well when the complexity of the modeled behavior and therefore the number of states and events (transitions) increases. Contrary to COP, combinations of orthogonal system variations need to be handled separately and each specific state and transition needs to be completely remodeled. This is mainly reasoned by the fact that FSMs do not provide many mechanisms of programming languages that help scaling them up, and that there is no support for a synchronization of multiple separate behaviors [1]. Furthermore, it is difficult to modularize certain behaviors since higher-level patterns that reoccur frequently cannot be captured. So it is often necessary to repeatedly rebuild similar behaviors from scratch [1].

Using COP in conjunction with state machines can help alleviate these problems to a certain degree. The number of states and transitions can be reduced, because orthogonal feature variations can be expressed with layers.

5. SUMMARY

In this paper, we motivated COP for the implementation of state machines in computer games. We presented ContextLua, our COP extension to the Lua language. Its implementation is based on the concept of metatables, which allows for a simple and elegant layer-aware method dispatch.

6. REFERENCES

[1] AiGameDev. 10 Reasons the Age of Finite State Machines is Over. `http://aigamedev.com/open/articles/fsm-age-is-over`.

[2] Malte Appeltauer, Robert Hirscheld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-based Software Composition. In *Proceedings of Software Composition 2010*, Lecture Notes in Computer Science. Springer-Verlag, June 2010.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-oriented Programming Languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM Press.

[4] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[5] Epic Games. UnrealScript Language Reference. `http://unreal.epicgames.com/UnrealScript.htm`.

[6] Fabien Fleutot. Metalua - Static Meta-Programming for Lua. `http://metalua.luaforge.net`.

[7] John David Funge. *Artificial Intelligence For Computer Games: An Introduction*. A. K. Peters, Ltd., Natick, MA, USA, 2004.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Boston, MA, USA, 1995.

[9] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[10] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic Contract Layers. In *25th Symposium on Applied Computing, Lausanne, Switzerland*, New York, NY, USA, 2010. ACM DL.

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The Design and Implementation of a Language for Extending Applications. In *Proceedings of XXI SEMISH (Brazilian Seminar on Software and Hardware)*, pages 273–284, Caxambu, 1994.

[12] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The Evolution of Lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, 2007.

[13] John E. Laird and Michael van Lent. Human-Level AI's Killer Application: Interactive Computer Games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 1171–1178. AAAI Press / The MIT Press, 2000.

[14] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented Programming: Beyond Layers. In *ICDL '07: Proceedings of the 2007 International Conference on Dynamic Languages*, volume 286 of *ACM International Conference Proceeding Series*, pages 143–156, New York, NY, USA, 2007. ACM Press.