Aspect Weaving for Multiple Video Game Engines using Composition Specifications

Ben J. Geisler[1] and Shane L. Kavage[2]
[1]*Saint Norbert College, DePere, WI, U.S.A.*
[2]*University of Wisconsin, LaCrosse, WI, U.S.A.*

Keywords: Aspects, Aspect-oriented-Programming, Video Game Engine, Gaming, Meta-language, DSL, Testing.

Abstract: In the realm of video game development, unique Domain Specific Languages (DSL's) are used in each of the most popular game engines making code sharing and reuse extremely difficult. For this reason, common software engineering practices such as design patterns and modularity have lagged. GAMESPECT is an aspect-oriented DSL (DSAL) that seeks to generalize concerns of video game programming. This paper explores the technology involved, namely composition specifications which enable the usage of XText and TXL to weave aspect code into multiple game engines and multiple languages. We describe the four main steps of the weaving process: reification, matching, ordering and mixing. Our results demonstrate the technical accuracy of the DSAL as well as the efficiency across several samples in Unreal Game Engine 4(UE4) and Unity. The DSAL employed is a single-to-many source language featuring transformation and aspect insertion (via weaving) to multiple languages in these engines including C++, Skookum Script, LUA, and C#. The GAMESPECT technology has been employed beneficially in modern video game development across active titles on the PC, Android and Nintendo Switch.

1 INTRODUCTION

Video Game Engines are complex pieces of software capable of 3D rendering, high end physics processing, particle effects, entity management, artificial intelligence, UI, player control and interaction, terrain transformations and internal game economies. The preceding list is only a portion of what game engines can do: modern engines have advanced fully capable simulation software in 3D worlds. For example, Unreal Engine 4 is routinely used for professional drone work, and cinema work (Martin, 2012). The other prominent game engine, Unity is used often for independent development. Together these two engines account for 80% of the game engine market, more than 2,400 games a year are created with these two pieces of software (Dillet, 2018) (Moby, 2019).

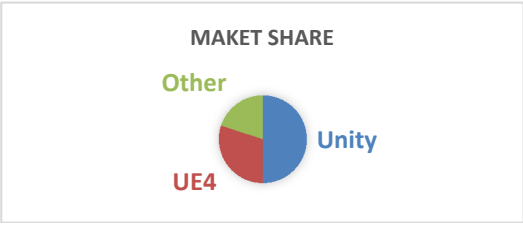

Figure 1: Game engine market share.

The two engines, as well as the other engines that exist such as Lumberyard and Crytek, all differ in the scripting language used. A scripting language is a DSL capable of specifying gameplay features unique to the game being developed (Anderson, 2001). Some researchers have begun cataloguing types of gameplay features, and with over 3,000 games existing each year, they have a lot in common (Moby, 2019) (Schell, 2019). Unfortunately, not only are the engines completely incompatible with each other, but the games typically have widely different design patterns (Boznjak & Orehovacki, 2019). This is problematic for many reasons. For one, no code can be shared between projects, even in the same organization. For two, common design principles are difficult to apply when the pattern might look widely different in different programming paradigms. These problems are only exacerbated when one realizes that internally each engine has a choice of scripting languages to use. It is intractable to suggest that code should be shared across most games, despite the fact that many games share common design patterns (Wang & Nordmark, 2015).

Aspect oriented programming was developed to help with cross-cutting concerns- the goal being that if a type of work is being done over and over across a codebase, the techniques used to solve this problem

Geisler, B. and Kavage, S.
Aspect Weaving for Multiple Video Game Engines using Composition Specifications.
DOI: 10.5220/0009444104540462
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 454-462
ISBN: 978-989-758-421-3
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

could be combined into "advice". Advice is code which can be called out at the time of each instance of the pattern (Kiczales et al., 1997). As mentioned above, many areas of code in a video game are shared (at least conceptually) between games even if the genres are vastly different (Holopainen & Bjork 2003). In particular, the three games we will focus our attention on are widely different. These video games are "shipped", modern production quality games: Noise Paradox, Rune 2 and Doctor Goose. Despite their differences in style and genre, many commonalities exist including spawning of enemies and tuning of player attacks.

GAMESPECT was a language developed as part of previous works, initially for Unreal Engine 4, by the authors and is now being used on the above three games (Geisler, 2019). This paper will explore one facet concerning the further development of GAMESPECT: the weaving process for a new engine: Unity. We will also discuss our process for evaluating effectiveness of our approach.

2 BACKGROUND

XAspects was one of the first meta-languages to incorporate the idea of a domain specific aspect-oriented language (Shonle, et al., 2003). The XAspects system makes use of a plugin framework which requires users to override a custom class called *AspectPlugin*, which resides in AspectJ. This class is ultimately the class which performs bytecode generation as applicable to whatever the original source languages were. Essentially, XAspects allows for integration of multiple domain specific languages into AspectJ. It does this by using AspectJ as a base language, and the aspects are defined above it. XAspects was created using trivial examples such as CD collection and referencing. The main idea was to use the aspect-oriented nature of AspectJ and have multiple DSLs translated into AspectJ (Shonle, et. al 2003).

The compositions specifications designed during the creation of the Awesome/SPECTACKLE meta-language (Kojarski & Lorenz, 2005) (Kojarski & Lorenz 2007) serve as inspiration for GAMESPECT. A composition specification is a middle layer between the source language and a set of plugin languages (in the spirit of XAspects). Multiple aspect-oriented languages can plug into one common aspect language with this approach (Lorenz & Mishali, 2012). All that needs to be done is specification of the translations needed to perform the code generation in AspectJ. Again, AspectJ is used as the base target language. The biggest difference between this approach and the approach of GAMESPECT is that Awesome goes from many languages to one language. Our approach goes from one language to many languages.

LARA is a recent meta-language approach which allows for one DSL type language to be translated to multiple target languages (Pinto et.al., 2018). The researchers of LARA have performed implementations in MATLAB, C and Java, to good results. Weaving is not automatic and is not based on any predefined rules or composition specifications. Instead, each language has a target translated: MANET is used for actual weaving in the C language. The methodology for which LARA has been evaluated is solid and can easily be used for comparisons; they have taken some examples of code and demonstrated lines of code savings and correctness (Pinto et.al., 2018).

GAMESPECT largely sits amongst the three methodologies above, using the idea of composition specifications from Lorenz et al, the idea of meta-languages from XAspects and the idea of multiple target languages from LARA (Kojarski & Lorenz, 2005) (Kojarski & Lorenz 2007) (Lorenz & Mishali 2012) (Shonle et.al, 2003). The biggest difference with GAMESPECT is the complexity of the composition specifications and the need for multiple platform support, which will be supported by modern language workbench tools. GAMESPECT is also an example of a one-to-many solution. The rest of this paper will describe the language, tools and framework.

3 METHODOLOGY

3.1 Composition Specifications

3.1.1 TXL Introduction

TXL is a source to source transformation language written by Dr. James Cordy (Cordy, 2006). TXL is a hybrid functional and rule based language that allows for pattern searching, unification and deep search via structural rewriting rules. Grammars are specified in *.grm* files, and the TXL processor implements these grammars, allowing for the source transformations.

In our approach, a composition specification can be thought of as a layer of abstraction sitting on top of TXL. TXL is the back-end of the source to source transformation system while the composition specifications serve as the front-end. At the front-end, we must provide a generic way for the transformation

to occur. In other words, if we require certain join points to occur (for example: *decHealth*), TXL must know how to replace the function decrement health with an aspect call to the appropriate advice. Furthermore, it must know how to do this in any of the available target languages.

Our approach to this problem is to templatize the TXL code needed for aspect weaving via *composition specifications*. To Lorenz et, al, a "composition specification" is a listing of which aspect languages to use, in which order and what parser to use (Lorenz & Mishali 2012). We generalize the idea of a composition specification such that it also includes the low level details of structure and syntax. GAMESPECT's composition specifications must include:

1) The explicit point cuts where advice should take place.
2) Any parameter passing or other syntax unique to these point cuts and this language.
3) A designator showing which language is being described.

3.1.2 Composition Specification Example

To illustrate composition specifications, one could consider the case of LUA scripting in UE4. LUA is an optional scripting language used by many Unreal Engine 4 developers for ease of syntax and implementation (Vasudevamurt & Uskov, 2015). The LUA language is implemented as calling hooks in a few places throughout the engine.

The three most popular points of insertion range from member functions to globals and static functions. All these calling sites must call out to the appropriate advice, which is why GAMESPECT implements composition specifications to dictate these sites. Providing that TXL supports the target language with a grammar, all it needs are the calling sites and the format of these sites. Grammars are provided to TXL via text files which describe a portion of the language grammar (Cordy, 2006). GAMESPECT provides a light DSL which sits between the user and TXL, generating TXL to be used for weaving from the composition specifications. GAMESPECT was originally written for Unreal Engine 4 and it's host of languages but as we'll see in this paper, adding the framework to Unity is as easy as providing composition specifications and any additional grammars needed by the language. Figure three shows one such set of composition specifications, in this case provided for LUA in Unreal Engine 4.

```
1   LUA BEFORE void UTableUtil::call( /* ) 1
2
3   LUA BEFORE int ue_lua_pcall(
4       lua State *L, int nargs, int nresults,
5       nt errfunc, char *func ) 5
6
7   LUA BEFORE void push_and_call(
8       char* top, int32 /?, int32 /?, FFrame& /? ) 1
9
10  LUA DURRING takeDamage( /* ) decHealth subtractHealth
```

Figure 2: LUA Composition Specifications for UE4.

3.2 Weaving Process

3.2.1 Overview

In aspect-oriented programming, advice code is the set of instructions which should be conditionally executed given a set of join points (a point cut). The join points determine when and where the code is allowed to execute (Kiczales et.al, 1997). For example, in the case of game engines- perhaps every time a "decrement health" function is called, we should multiply the incoming parameter by an amount which reflects the difficulty level of the game.

Weaving is the process of inserting advice code at certain join points inside the original code. The output of the weaving process is a new set of source code or binary code which performs the original code as modified by the aspect code (Courbis et.al, 2005). For example, consider figure four: the conditional code which says "if difficulty level is hard, multiply damage by two" is now inserted into the original code and the game runs with this new set of instructions.

```
1   Aspect AIDamage (Events)
2   {
3       declare Pointcut BEFORE decHealth(
4           float damage, Entity e)
5       {
6           if(Skookum.GetDifficulty() == "Hard")
7           {
8               e.takeDamageInternal(damage);
9           }
10      }
11  }
```

Figure 3: Aspect file for damage to an enemy.

GAMESPECT must provide for the means to find join points in source code. Unfortunately, this not a trivial task- the source language can change. Unreal Engine supports four or more scripting languages and Unity supports at least three.

Furthermore, each engine's scripting languages are custom and even if Unreal Engine 4 supported C#, it would be a slightly different grammar than supported by Unity. This means that composition specifications (as described in section 3.1.2) combined with TXL grammars must be used to find the appropriate join points. Also, the matter of finding

join points is non-trivial. While TXL is very good at source to source transformation, it is not a complete grammar of a full language implementation. In some cases, most the language has been captured by TXL, but especially in the case of C++ this is not true. Instead we turn to CPPAST which is a wrapper around the popular Clang compiler (Duffy et.al, 2014). CPPAST allows us to traverse the AST of C++ to find the appropriate call sites during the weaving process.

The second area in which GAMESPECT is useful is the area of the actual advice code which is to be woven. In the case of game engines, it may be necessary to do more than simply adjust an integer value. It might be necessary to detect boss names or check for additional conditions based on values. Essentially, an entire DSL is needed to describe the changes to be made. The GAMESPECT framework provides a scripting language which is then compiled down to a given target language (in the case of UE4- Skookum Script is used, in the case of Unity- C# is used) (Geisler, 2019).

To fully understand the weaving process, first let's consider what the woven code will look like. The woven code should incorporate conditional advice code across the applicable functions. In other words: if one of the functions from the composition specification is called, then check function names registered in the advice. If there is a match, insert the advice code. Considering an example will make it easier to describe the process by which they are woven. As an example, let's consider a decHealth method which is being called: we'd like to adjust the damage based on easy or hard difficulty modes. Perhaps on hard mode, the damage is doubled.

First, we need to determine where decHealth is being called from, theoretically it can be called from anywhere in the codebase but in practice there are only three possible call sites (shown in figure three) in UE4. The three ways of making LUA function calls in Unreal Engine 4 have slight differences. The first is a class member, the second is C-style and the third is static. In addition, the parameters are of different types, names and numbers. The TXL rules generated by GAMESPECT must be slightly different for each. For example, in figure 4 we can see the TXL rules for the class member version of the LUA call.

Once the TXL rule specifications are generated, we will run the Clang tools with the function list from the aspect file, and find the applicable files on which to run the TXL files. Once these are found, TXL is run and every call site refers to an instance of the target language which has the advice code to run.

XText is used on the GAMESPEECT advice code to generate runtime compatible Skookum Script code (in the case of Unreal Engine 4) or C# code (in the case of Unity). Figure four shows the code that in our example would modify the damage by a multiple of two, essentially calling TakeDamage one additional time if the gameplay difficulty is set at "Hard". XText must be provided a full grammar for GAMESPECT, this is the subject of other papers (Geisler, 2019) and outside the scope of this discussion.

The code in figure four is transformed to skookum script code by XText as part of the GAMESPECT parser. XText is a modern DSL workbench (Eysholdt & Berens, 2010). XText is provided a full set of code generation rules. In doing so, the code which is generated can be called "on the fly" by the calling site code. In other words, if LUA calls decHealth, since it was woven- it will call Skookum and the advice will be performed.

3.2.2 Reification

GAMESPECT employs a similar version of weaving to that which was researched by Kojarski and Lorenz in their works (Kojarski & Lorenz, 2005) (Kojarski & Lorenz 2007). The abstract weaving process is composed of four sub processes: reify, match and order/mix. In reification we take a calling specification and construct a weaver representation of that class. The specification generation should include all computation "shadows" which occur throughout the codebase. Intuitively, these shadows are all the locations of the calls which should be advised upon: reification lists all possible spots where the join points can occur.

Reification in GAMESPECT either uses CPPAST/Clang (for C++ based codebases) or TXL (for non-C++ codebases). For Unity we will use the C# grammar of TXL and run every file through TXL with a rule that extracts function names.

The matter is a little more complicated when using C++ engines since TXL does not include a full C++ grammar and many customizations exist to most C++ grammars, including Unreal Engine 4.

For this reason, GAMESPECT uses CPPAST and Clang based tools. CPPAST was built upon Clang which is an LLVM parser for C++ and can easily be customized to support the UE4 codebase. There have been a couple source to source transformation languages built in recent years around Clang, CPPAST is one of them (Antao et al., 2016) (Duffy et al., 2014).

CPPAST allows for extrapolation of function names from headers. Internally the software uses

```
rule addCallToUTableUtil
        construct JoinPointFunctionIDs [repeat id]
                'UTableUtil::call
        replace $ [function_definition]
                Specifiers decl_specifiers DeclaredItem [declared_item]
                Extensions [repeat declarator_extension+] CtorInitializer [opt ctor_initializer]
                Exceptions [opt exception_specification]
                Body [function_body]
                        deconstruct * [id] DeclaredItem
                                FunctionId [id]
                        deconstruct * [id] InterestingFunctionIds
                                FunctionId
                        construct StringFunctionId [stringlit]
                                _ [+ FunctionId]
                        by
                                Specifiers Pointer DeclaredItem Extensions
                                CtorInitializer Exceptions
                                Body [addCall StringFunctionId] end rule
```

Figure 4: addCallToTableUtile for UTableUtil::call : LUA join point example for GAMESPECT.

libclang and provides an AST for C++ code. We then write a simple traversal of the AST as part of the GAMESPECT glue code (API).

3.2.3 Matching

Matching takes the results of reification and calls TXL on each of the files which holds the named functions. While the composition specification is used for reification, the GAMESPECT aspect script (figure 3) is used for exact function header matches during this phase. Asterisks in the GAMESPECT script signify one or more arguments of any type. The non-presence of variable names implies that no parameters were present on the function. Via rule pattern matching, the exact matches for hits will be conducted. For example, takeDamage is described as follows in the aspect script:

```
declare Pointcut BEFORE
   takeDamage(float damage, Entity
   *e) decHealth subtractHEalth
```

In this case, takeDamage takes a float and Entity pointer. Only the appropriately labeled functions in the codebase will be modified since TXL allows for these parameters. Since the composition specification lists the names of the functions to be found, these must correspond to named functions in the aspect script (figure 3). In our work "matching" corresponds only to the part of weaving which actually finds the call sites.

For each of the call sites (e.g. decHealth from figure 3), if that function is called during runtime, we want the corresponding advice to be called.

Therefore, during load time of the game, matching is called. GAMESPECT is initialized at this time and both the composition specifications as well as all available aspect files are interpreted. Upon finding a valid composition specification, if there is a join point named after the line definition, a key/value pair was added to the hash matcher. Note that if there isn't a join point named after the line, it means it's a generic call and therefore anything which is the named function is called with is a potential calling site. The parameter used for lookup is numbered in the calling specification and recorded in the Hash Matcher class, such that when it is called with all its parameters, it will use that ordering to call the skookum script aspect which was generated earlier by GAMESPECT. For example, in "push and call" of figure 2, the first parameter is used to get the correct function name to be called for pointcuts. If *onbeat* is to have a point cut, then *onbeat* must be registered (figure 5).

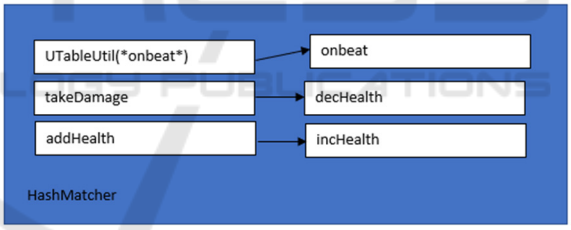

Figure 5: Hash Matcher mapping diagram.

4 RESULTS

GAMESPECT has been used and tested on a few recent video games: Noise Paradox, Rune 2, and Doctor Goose. One of the games using GAMESPECT is Noise Paradox: it is available for beta testing on some android storefronts. This trio of games demonstrates that GAMESPECT is genre-agnostic, since each of the games is a different genre. Also, one of the games is primarily Unity based and the others are Unreal Engine 4. But yet, all of them can use GAMESPECT. Likewise, there is a good variety in scripting languages chosen: from LUA and C++ to Skookum Script and C#, most languages are represented.

GAMESPECT was tested on Rune 2 and a custom version of Noise Paradox which was programmed in Unity and Unreal Engine 4, as separate versions. Several segments of code were tested, all of which represent example game balance tasks. LUA, C++, Skookum Script and C# (Unity) were used. Also, comparisons were drawn between the efficiency of GAMESPECT in UE4 vs. the efficiency in Unity. For the solution to be tractable to game developers, this framework needed to run at a solid 30FPS with no noticeable performance impacts. It should be noted that Noise Paradox is a beat matching game similar to Rock Band or Guitar Hero (Miller, 2009). This means that many of the tasks are beat or music oriented because the player needs to match gameplay actions to the beat.

Ten tasks (functions) were monitored and optimized by use of GAMESPECT (aspects) and compared also to their *non-aspect* versions.

4.1 Preliminary Results

The general process for testing was to write the gameplay functions in traditional object-oriented-programming and then to use aspect-oriented-programming via GAMESPECT. We used Visual Studio Assist to find applicable join points, and pull out aspects, putting them into the GAMESPECT framework. There were two main concerns with using GAMESPECT which we wanted to test: correctness and efficiency. Correctness was demonstrated by functionality not changing in the final product. For a period of time, two separate versions of each game were passed through quality assurance, with bugs being reported. The end products reached "zero bug regression" on both versions (OOP and AOP) which is the typical gold standard for bug finding (Gu et al., 2010). The deviation of aspect-oriented vs. object oriented is almost non-existent, as figure 6 shows.

Pulling common code into aspects has the effect of reducing the total code used at the join point, assuming it is used more than once. We wanted to know if there is indeed a lines of code savings for pulling out aspects. In theory since multiple call sites exist for each piece of advice, there should be a line

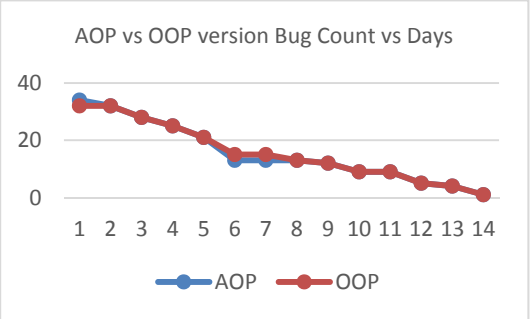

Figure 6: The plot of bugs starting with two weeks remaining on the project.

of code savings, which correlates to easier maintenance (Polo et al., 2001). This can be verified in tables one and two. The range of savings varied from around 9% to 40%. It should be noted that any line of code savings above 10% is considered substantial for maintenance purposes (Polo et al., 2001). It would be unfair to average together the savings since some functions are called more often than others at run time, so it's hard to say in practice how many lines of execution are saved. However, according to Pinto et al, LARA had efficiencies ranging from 10-20% (Pinto et al., 2018). GAMESPECT is easily in that range, and sometimes better, topping out at 68%.

Given the extra calls made at runtime it was also necessary to gauge performance. Therefore, we also ran a runtime analysis to ensure there was no performance impact (figure 7). Similar to LARA, GAMESPECT has virtually no performance impact.

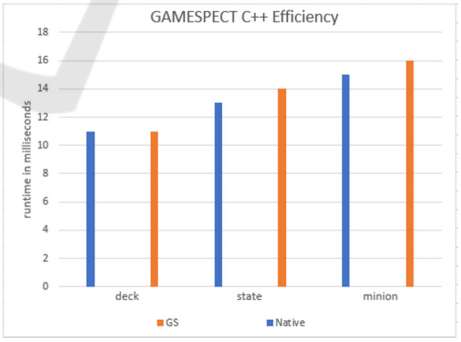

Figure 7: Test Suite 2 CPP runtime efficiency.

Table 1: LOC Savings for Various Functions in Test Suite 2 (UE4).

	original lines	original files	total lines	GAMESPECT lines	Destination Lines Total	Savings in Lines	Savings Percent
AIStateAdvancing (CPP)	10	2	20	3	17	3	15%
PlayerStateAdvancing (CPP)	14	3	42	5	26	16	38%
LevelStateAdvancing (CPP)	8	5	40	8	13	27	68%
SyncTracks (SK)	5	3	15	4	13	2	13%
FastFoward (SK)	4	3	12	2	5	7	58%
Rewind (SK)	4	3	12	2	5	7	58%

4.2 Unity Test Results

The creation of an entirely separate code project for Noise Paradox involved extensive person-hours and work. However, we believe that it's the only way to effectively ensure that GAMESPECT is truly generic enough to run on multiple game engines. Creation of the Unity version occurred over the later months of 2019 and is ongoing while the Unreal Engine 4 version is completed. Some files were ultimately different than the others, but the design for the test suite functions remained consistent across both UE4 and Unity. We would like to see very little difference between the Unity version of Noise Paradox and the C++/Skookum version. In fact total savings should be similar (albeit negligible due to language differences).

To understand if this claim is true, we tested all the same test suite functions and files on a version of Noise Paradox which runs Unity (C#). The data is fairly consistent. We'd expect this since C# and C++ are not very different in terms of syntax. The differences are largely due to coding style and initialization differences. This is shown in figure 8.

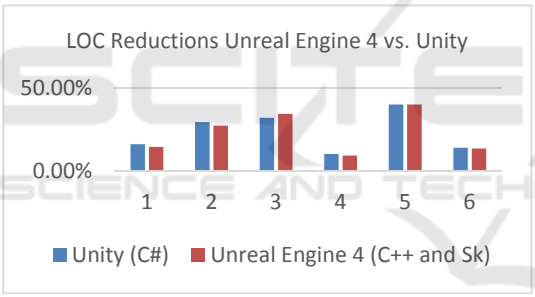

Figure 8: Efficiency comparison for GAMESPECT.

5 CONCLUSIONS AND FUTURE WORK

There are multiple impacts of this research for a few different concerns including academic, professional and gameplay theoretical.

The above tests, written in both traditional (OOP) programming and aspect oriented GAMESPECT, show a clear advantage to using Aspect Oriented Programming. Also, from the academic perspective, we have demonstrated that a one-to-many source to source transformation DSL is possible provided composition specifications. Furthermore, we have demonstrated that the insertion of such a mechanism is pluggable. Previous to this work, LARA demonstrated that a one-to-one translation was certainly possible (Kojarski & Lorenz, 2005),. Likewise, Awesome and SPECTACKLE provided a many-to-one solution (Kojarski & Lorenz, 2007),. As far as we know this is the only contribution with a one-to-many solution which uses an intermediate composition specification scheme.

On the professional front, the benefits are numerous. The primary author has been contacted by several game studios to use GAMESPECT in their commercial large-scale endeavors. The author has also presented the GAMESPECT framework and code at a couple game conferences with fantastic feedback. The main reason studios are wanting to use GAMESPECT is that their designers often code in more than one language. Meanwhile their producers can be left out of the loop in terms of tuning certain parameters- especially when it comes to game balance.

The final significance of our research is that game balancing principles have finally become tangible with GAMESPECT. For many years, researchers have written about topics such as MDA (Mechanics, Dynamics and Aesthetics) (Hunicke et al., 2004). Other gameplay designers such as Schell would write about rules for balancing games (Schell, 2019). But very few researchers have provided a cohesive framework- and when they did it was just for one game/architecture (Dormans, 2012).

5.1 Future Work

The selection of an appropriate TXL template given a certain calling specification is a process which requires a language enumeration to be set. This means one specification can only target one language. However, in the case of languages with similar features, this may be inefficient. In the future this could be an automatic process which takes the calling specification and determines the appropriate language outputs for each target language. Matching the calling specifications to applicable target languages would be an interesting topic in and of itself and may feature running TXL multiple times once against every instance.

Table 2: C# Lines of Code Savings.

	total lines	GAMESPECT lines	Destination Lines Total	Savings	
beatNow (C#)	36	3	30	6	16.00%
calculateBPM (C#)	75	5	53	22	29.33%
spawnAtPlayer	75	6	41	24	32.00%
CharacterDeckComponent (C#)	50	3	45	5	10.00%
StateManagerComponent (C#)	10	1	6	4	40.00%
MinionController (C#)	29	1	25	4	13.79%

In the future, there is no need to continue writing code generators for each engine used. While UE4 uses Skookum Script, this is not available in Unity and hence a code generator for C# was needed. It would be an enhancement to combine these two initiatives in one and use a shared code generation target. LUA would actually be a good choice for the target language since it is supported by Unity as well as Unreal Engine 4. Furthermore, it has a fairly small footprint (Glasberg & Bresler, 2006) such that even if a target game engine doesn't support LUA, it could easily be added.

The inclusion of generic function calls such as "push and call" from figure 2 allows for LUA to work in an aspect-oriented fashion on any created LUA script as long as it's registered in the .gs file. This is an attractive and extensible solution since theoretically one could add Blueprint support very easily. The CPP composition specification would only need to register the C++ call sites which are used for Blueprints and this would work. Due to time constraints, the current research has not included Blueprints, however they are very popular in UE4 development (Wang & Nordmark, 2015). Adding support for these would be critical to widespread acceptance.

REFERENCES

Antao, S.G., et al. 2016 Offloading support for OpenMP in Clang and LLVM. In: *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC. IEEE Press*.

Bosnjak, M. & Orehovacki T.. (2018, May). Measuring quality of an indie game developed using unity framework. *In 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (pp. 1574-1579).* IEEE.C. J. Kaufman, Rocky Mountain Research Lab., Boulder, CO, private communication, May 1995.

Anderson, E.G., 2001. A Classification of Scripting Systems for Entertainment and Serious Computer Games. In *2011 Third International Conference on Games and Virtual Worlds for Serious Applications*

Cordy, J.R., 2006. Source transformation, analysis and generation in TXL. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (pp. 1-11). ACM.*

Courbis, C. Carine, &Finkelsteiin A., 2005 Towards aspect weaving applications.In *Proceedings of the 27th international conference on Software engineering. ACM*

Dillet R., 2018. "Unity CEO Says Half of All Games Are Made on Unity," *https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/.*

Dormans,J., 2012, Engineering emergence: applied theory for game design. *Universiteit van Amsterdam Press*

Duffy, E.B., Malloy, B.A., Schaub, S., 2014. Exploiting the Clang AST for analysis of C++ applications. In *Proceedings of the 52nd annual ACM southeast conference.*

M. Eysholdt, M., Behrens H., 2010. Xtext: implement your language faster than the quick and dirty way. IN *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM,*

Hunicke, R. LeBlanc, M., Zubek.R, 2004. MDA: *A formal approach to game design and game research. In Proceedings of the AAAI Workshop on Challenges in Game AI. Vol. 4*. 2004.

Geisler B.J., 2019. *GAMESPECT: A Composition Framework and Meta-Level Domain Specific Aspect Language for Unreal Engine* 4.NSUWorks Nova Southeastern

Glasberg M., & Bresler J., 2006. The Lua Architecture. In *Advanced Topics in Software Engineering*.

Gu,Z., et al. 2010. Has the bug really been fixed?,In *ACM/IEEE 32nd International Conference on Software Engineering IEEE*.

Holopainen, J., & Björk S. (2003). Game design patterns. In *Lecture Notes for GDC.*

Kiczales., G., et al. 1997. Aspect-oriented programming. In *European conference on object-oriented programming (pp. 220-242). Springer, Berlin, Heidelberg*.

Kojarski, & D.H. Lorenz, D. H. (2007). Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. ACM Sigplan Notices, 42(10), 515-534.

Kojarski, S., & Lorenz, D. H. (2005). Pluggable AOP: Designing aspect mechanisms for third-party composition. *ACM SIGPLAN*

Lorenz, D. H., & Mishali, O. (2012, March). SPECTACKLE: toward a specification-based DSAL composition process. In *Proceedings of the seventh workshop on Domain-Specific Aspect Languages*

Martin M. (2012). The Technology behind the Unreal Engine 4 Elemental Demo. *Part of "Advances in Real-Time Rendering in 3D Graphics and Games," SIGGRAPH, 2012.*

Miller, K. (2009). Schizophonic performance: Guitar hero, rock band, and virtual virtuosity. *Journal of the Society for American Music*, *3*(4), 395-429.

Moby Games Stats," n.d. https://www.mobygames.com/.

Pinto, P., Carvalho, T., Bispo, J., Ramalho, M. A., & Cardoso, J. M. (2018). Aspect composition for multiple target languages using LARA. *Computer Languages, Systems & Structures*

Polo, M., Piattini, M., & Ruiz, F. (2001, November). Using code metrics to predict maintenance of legacy programs: A case study. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001* (pp. 202-208). IEEE.

Schell, J. (2019). *The Art of Game Design: A book of lenses*. AK Peters/CRC Press..

Shonle, M., Lieberherr, K., & Shah, A. (2003, October). XAspects: an extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*.

Vasudevamurt, V. B., & Uskov, A. (2015, May). Serious game engines: Analysis and applications. In *2015 IEEE International Conference on Electro/Information Technology (EIT)*

Wang, A. I., & Nordmark, N. (2015, September). Software architectures and the creative processes in game development. In *International Conference on Entertainment Computing* (pp. 272-285). Springer, Cham

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS