LARA as a Language-Independent Aspect-Oriented Programming Approach

Pedro Pinto
Faculty of Engineering
University of Porto
p.pinto@fe.up.pt

Tiago Carvalho
Faculty of Engineering
University of Porto
t.carvalho@fe.up.pt

João Bispo
Faculty of Engineering
University of Porto
jbispo@fe.up.pt

João M. P. Cardoso
Faculty of Engineering
University of Porto
jmpc@fe.up.pt

ABSTRACT

Usually, Aspect-Oriented Programming (AOP) languages are an extension of a specific target language (e.g., AspectJ for Java and AspectC++ for C++). This coupling can impose drawbacks such as arbitrary limitations to the aspect language. LARA is a DSL for source-to-source transformations inspired by AOP concepts, and has been designed to be independent of the target language. In this paper we propose techniques to overcome some of the challenges presented by a language-independent approach to source code transformations, and present and discuss possible solutions and their impact. Additionally, we present some of the benefits and opportunities of this approach. We present an evaluation of our approach, show that we can significantly reduce the effort to develop weavers for new target languages and that the proposed techniques contribute to more concise LARA aspects and safer semantics.

CCS Concepts

•Software and its engineering → Compilers; Source code generation; Domain specific languages;

Keywords

LARA; Aspect-Oriented Programming; Modularity

1. INTRODUCTION

Aspect-Oriented Programming (AOP) is a paradigm that aims at increasing program modularity by specifying code related with crosscutting concerns (e.g., logging, profiling, autotuning) into separate entities called aspects.

SAC 2017,April 03 - 07, 2017, Marrakesh, Morocco

http://dx.doi.org/10.1145/3019612.3019749

It is common for an AOP approach to be focused on a specific target language (e.g., AspectJ [13] for Java and AspectC++ [21] for C++). When moving crosscutting concerns to an aspect, we are effectively moving code from a source file to an aspect file. So for seamless integration we need access and support to language features. In fact, most traditional AOP languages are extensions of their target language [10].

Despite the benefits provided by such approaches, coupling an AOP language to a target language can present drawbacks (e.g., restrictions in the AOP model due to target language limitations) and prevent opportunities (e.g., tooling reuse between AOP approaches of different target languages).

LARA [4] is a domain-specific language (DSL) for source-to-source transformations and analysis, inspired by AOP concepts [12]. LARA explores the idea that it is possible to have a single, target-independent language capable of expressing source-code transformations for any language. However, this approach poses some challenges, namely querying specific program points in a language-independent fashion and specifying additional behavior on the target program. In this paper we propose techniques that allow to overcome the challenges of this approach, as well as present some of its benefits.

The remainder of this paper is organized as follows. In Section 2 we explain our motivation for a language-independent AOP approach, its opportunities and challenges. Section 3 briefly introduces LARA and presents techniques to overcome some of the challenges of this approach. Section 4 discusses the results obtained by applying the techniques in this paper. Section 5 presents the related work, and Section 6 concludes the paper.

2. MOTIVATION

Common AOP approaches usually use aspect-definition languages that are an extension of the target language (e.g., AspectJ [13], AspectC++ [21], AspectMatlab [1]). The most relevant benefits from this approach is being able to specify additional behavior transparently in the aspect language,

and a possibly lower learning curve for aspect developers, since they should be familiar with the target language.

However, tying an aspect language to its target language has at least two drawbacks. First, extending the target language can impose arbitrary limitations on the aspect language itself (e.g., since AspectC++ [21] extends C++, it inherits its limited reflection capabilities). Second, developing an aspect-oriented approach for a new language from the ground up is non-trivial and a significant undertaking, in part because the common parts between AOP approaches that can be reused are few or non-existent.

We propose that these two problems can be solved, or significantly reduced if we adopt an AOP approach which uses a single Domain Specific Language (DSL) agnostic to the target language. We consider that, on one hand, it allows the development of an AOP language with features that are independent of the target language; on the other hand, it enables tooling reuse, which can significantly reduce the effort needed to support new target languages. This can include compilers and/or interpreters for the DSL in addition to a well-defined API. There are also new opportunities to explore, such as the possibility to reuse aspect code between different target languages.

An aspect-oriented approach that is agnostic to the target language presents new challenges. We consider that two of the main challenges of a language-agnostic approach are how to specify, in a target-independent way, 1) queries of specific points in the code and 2) additional behavior for the target language. Previous work on the LARA language [4, 5] provides a solution for the first challenge, which is briefly presented in Section 3.1.

Regarding the second challenge, one of the ways LARA currently deals with it is to allow insertions of arbitrary strings around the points of interest it captures. Figure 1 shows a LARA aspect that modifies source code to log function calls. Line 3 queries the code and selects all calls inside functions (i.e., `select function.call end`). The join point `function` represents the code for a function while the join point `call` represents the code a function call. To this selection, we apply the rule inside the `apply` block (lines 4-8), which inserts the code inside the brackets (i.e., `%{}%`) before all captured calls. The inserted code prints the name of function where the call happened (i.e., `$function.name`), and of the called function (i.e., `$call.name`).

Although a powerful mechanism, raw code insertion can also become complex and error prone. For instance, syntax verification of the inserted code during aspect compilation is not guaranteed and is dependent on the implementation of the tool. Also, this example is not inserting the necessary includes (`<stdio.h>`) for the function `printf`. In this paper we build upon our current solution and present techniques to better overcome this second challenge.

3. OUR APPROACH

In this section we describe our approach, starting by briefly describing the LARA language and then explaining the techniques we use, LARA libraries, generic weaver actions and join point aliases.

```
1  aspectdef LogCall
2
3   select function.call end
4   apply
5    $call.insert before %{
6     printf("[[$function.name]]->[[$call.name]]\n");
7    }%;
8   end
9
10 end
```

Figure 1: An example of a LARA aspect that inserts a 'print' instruction before function calls.

3.1 The LARA Language

LARA [4] is a domain-specific language for source-to-source transformations and analysis, inspired by AOP concepts [12]. LARA provides semantics that allow to query and modify points of interest in the target source code, and supports arbitrary JavaScript code to provide general-purpose computation.

Unlike most AOP approaches, the authors of LARA designed it to be independent from the target language and to be used to transform any kind of target code. This was achieved by decoupling LARA from the specification of the points of interest of the target language (i.e., the Language Specification). When using LARA code to transform a specific target language we need to build a *weaver*, which connects the language specification and the target code representation, e.g., an Abstract Syntax Tree (AST).

Syntactically, LARA aspects have three main keywords, `select`, `apply` and `exec`. For more detailed information please refer to previous work [4, 5]. The first two can be seen in the aspect presented in Figure 1 and they are used to *select* points of interest in the code and *apply* actions over them. The `exec` keyword is used inside `apply` blocks to call a weaver-specific action over a join point. LARA has two default actions, `def` and `insert`, which have their own specific syntax and are used to define the value of a join point attribute and to inject code in the program, respectively. All other actions are added by the weaver developer and are called with the `exec` keyword. An example can be seen in Figure 5, where the `DeclareVariable` action is used.

Figure 2 shows the high-level structure of MANET, a LARA weaver for the C language. The LARA engine, which is generic and can be used by all LARA weavers, contains a compiler and interpreter for the LARA language. It needs a language specification (e.g., *C Lang. Specs* in the figure) to know which points in the code are available (*join points*, in AOP jargon [12]), and a Weaving Engine that performs the translation between LARA points (e.g., function, loop) and the real points in the source-code representation (e.g., Procedure, ForLoop). In this case, the Weaving Engine uses Cetus [7], a tool that already provides an AST for C code. The Language Specification does not need to provide a 1:1 mapping to the target language, only the points of interest. It is possible to have a Language Specification that, for instance, only captures the loops (and associated information) of a target language.

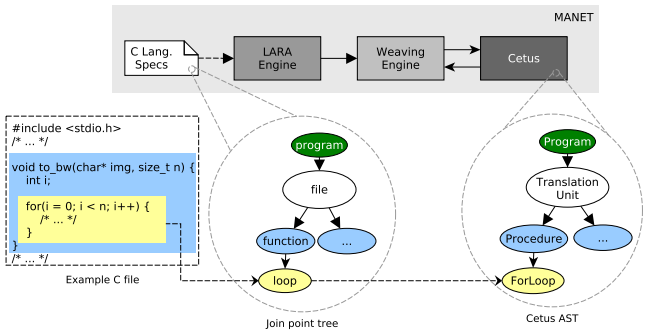

Figure 2: Block diagram of MANET, a LARA-based tool for C source-to-source transformations.

LARA considers two separate groups of developers: the weaver developers and the aspect developers. This separation is similar to "Stephanies" and "Joes" introduced by Pingali [2]. The weaver-developers ("Stephanies") are few, but their work enable AOP approaches for new languages that can be used by many aspect-developers ("Joes"). Additionally, the effort spent on the side of the weaver-developer has the potential to multiply its payoff, in proportion to the number of users of the weaver.

Several techniques presented in the next sections (e.g., libraries) take this into account, and consider that their implementation effort need to be done only once per weaver.

3.2 Libraries

LARA was developed as a modular AOP approach and supports importing code, such as libraries. LARA supports two kinds of libraries, Javascript and LARA. Javascript libraries are standard Javascript code, that can be called anywhere in a LARA aspect. LARA libraries are composed of LARA aspects that in addition to what Javascript libraries do, can also perform queries and transformations over the target code.

Javascript libraries are useful for developing utility libraries that deal with code generation. Consider the example in Figure 1. If we have a Javascript library `CodeGen` that provides the function `println` that returns the code for printing a line, lines 5-7 can be replaced with:

```
var code = CodeGen.println(
    "[[$function.name]]->[[$call.name]]");
$call.insert before '[[code]]';
```

Although there is little improvement from the previous version, this technique is used as a stepping stone for more complex examples.

LARA libraries are used when there are common tasks that either query or modify the target code. Examples include monitoring techniques such as timing parts of the program (e.g., function and loop execution) and logging activities such as error conditions and function calls.

While aspect developers can use these facilities to write better structured code, weaver developers can take the most advantage from it by providing APIs that hide complexity from the aspect developer. Figure 3 shows a LARA aspect that uses a Timer LARA library that can be used to perform simple timing measures of points in the code. Line 1 imports the library, and line 5 creates an instance of a timer aspect. Line 7 selects all the loops inside the function `main` and line 9 uses the timer aspect to insert a timing measure around the loop and a print of the result. Multiple weavers can provide their own implementation of this LARA library for their target language. This results in a generic library, which allows aspect developers to write strategies at a higher abstraction level while also being closer to language-independent aspects. The LARA aspect presented in Figure 3 can be used to weave source code from multiple target languages, as long as their weavers implement their version of the Timer library.

For an example input program, a C weaver would generate the code in Figure 4. The program now measures and prints the execution time of all loops in the main function, using a C library (`Timer.h`) provided by the weaver-developer. This specific Timer LARA library exposes a number of other functions for a more fine grained control over the timing features. For instance, there are functions to insert start, pause and stop calls, which can be used to time arbitrary parts of a program. Instead of directly printing the value of the timer, there is a library function to store the value in a named variable, which can be used in a more elaborated print message, or stored in a file.

```
1 import lara.inst.Timer;
2 aspectdef main
3  select function{"main"}.loop end
4  apply
5   call timer : NewTimer("timer_" + $loop.rank);
6   timer.aroundAndPrint($loop);
7  end
8 end
```

Figure 3: LARA aspect using a Timer library to measure loop execution time in the `main` function.

```
1  #include <stdio.h>
2  #include "Timer.h"
3  Timer* tim1 = NULL;
4  int main()
5  {
6      int i, a = 42;
7      timer_init( tim1 );
8      timer_start( tim1 );
9      for ( i = 0; i < 100; i ++ ) {
10         i = ( i + a );
11     }
12     timer_stop( tim1 );
13     timer_print( tim1 );
14     tim1 = timer_destroy( tim1 );
15     return 0;
16 }
```

Figure 4: The resulting C code when applying the aspect presented in Figure 3. We highlight the code inserted for setup and timing and printing calls.

```
aspectdef DeclareVariableInLoop
 select loop end
 apply
  exec DeclareVariable(Type.FLOAT,'X'+$loop.rank,3);
 end
 condition
  /*...*/
 end
end
```

Figure 5: An example of how a LARA action can be used to perform actions such as declaring variables.

A further step is to standardize libraries between weavers. Since libraries can be developed at the weaver level, if two weavers share the same API (e.g., `lara.inst.Timer`) we can enable aspect code compatibility between weavers, at the library level. If besides the used APIs, the weavers also share the part of the Language Specification that is used in an aspect (e.g., *join points* `function` and `loop`, in Figure 3), the aspect can be fully compatible between weavers, even in cases where the target language is not the same.

3.3 Generic Weaver Actions

Since weavers have access to the complete AST of the original program, certain source code transformations (e.g., loop unrolling, adding C includes) might be easier to implement in the weaver itself than within LARA aspects. The language specification allows the definition of custom actions, that are implemented by the weaver developer. For instance, consider the case where we intend to declare a variable inside a given scope. Depending on the language, there can be several syntactic and semantic rules associated with this action. While this could be done with insertions of native code, a weaver action provides greater control and safety (e.g., check whether there is a variable with the same name in the given scope, update the symbol table, warn the user if the declaration shadows another variable, etc). Figure 5 shows an example of a LARA aspect that uses a weaver action to declare a variable inside the scope of all the loops in the given code.

If several weavers conform to the same standard and implement an API with the same actions providing the same semantics, we can say we have generic weaver actions, even if such actions are considered weaver specific, as mentioned in Section 3.1. For instance, if two weavers, one for Java and one for C, both implement the `DeclareVariable` action (seen in Figure 5) for their target language, with the same interface and semantics, it is considered a generic action. These actions improve the development of language-independent aspects using LARA.

3.4 Join Point Aliases

In certain cases, there are points of interest in the code that are similar between languages, but that can have different names due to the history of the language and conventions (e.g., `function` in C vs `method` in Java). To increase compatibility between weavers, the language specification supports the definition of *join point aliases*, which allows referring to the same join point using different names. Instead of forcing a single denomination to all languages, weavers can use their "natural" denomination and still have compatibility with more generic aspects.

For instance, in Kadabra, a Java weaver, it is possible to capture methods with the following code:

```
// these select statements are equivalent
// because function is an alias for method
select method end
select function end
```

4. EVALUATION

In this section we evaluate the approach using three weavers that target different languages: Kadabra[1] for Java, MANET[2] for C and MATISSE[3] for MATLAB.

4.1 Tooling Reuse

One of the objectives of this approach was to enable tooling reuse between weavers that target different languages, using LARA as their aspect language. We assume that when developing a new weaver, developers will most likely reuse existing grammars, parsers and ASTs for the target language, and this should not count towards the programming effort. For instance, both MANET and Kadabra use third-party compiler frameworks (Cetus [7] and Spoon [17], respectively) to parse the code and obtain the AST. MATISSE reuses a custom parser and AST that was originally developed to translate MATLAB to C.

The LARA Framework, written in Java, contains a LARA compiler, a LARA interpreter and a Weaver Generator. The compiler parses LARA aspects and creates an intermediate representation in XML which can be executed by the LARA interpreter. The Weaver Generator is a tool that accepts a Language Specification and generates a skeleton weaver for that specification. The generated implementation already includes the LARA compiler and the LARA interpreter, and can be immediately executed. The task of the weaver developer is to fill in the blanks and write the code that connects the points in the specification to the nodes in the AST of the source code. Using the Weaver Generator, and having a parser and AST for the target language, it is possible to have a working prototype in a few hours.

Figure 6 shows the logical source lines of code (L-SLOC)[4], for the Kadabra, MANET and MATISSE weavers, divided into four components. *LARA* is the code size of the LARA framework, which is shared by all weavers. This represents a large part of the total code that takes care of compiling and interpreting LARA aspects. The *Generated* slice represents LARA API code that is automatically generated by a tool of the framework. This is the interface between a weaver implementation and the LARA framework and consists mainly of abstract classes that the weaver developer needs to implement. *Compiler* accounts for the code of the source-to-source compiler used by each weaver, Spoon for

[1]https://specs.fe.up.pt/tools/kadabra
[2]https://specs.fe.up.pt/tools/manet
[3]https://specs.fe.up.pt/tools/matisse
[4]Provided by LocMetrics: http://www.locmetrics.com/

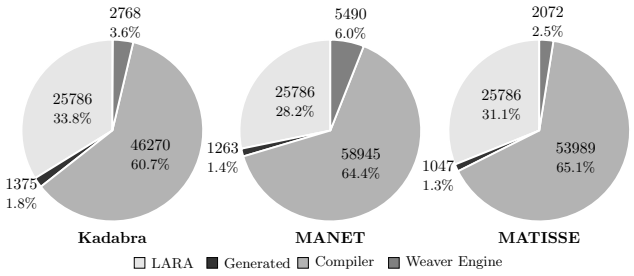

Figure 6: Lines of code for each weaver divided into components: LARA framework, auto-generated API, weaver engine and compiler framework.

Kadabra, Cetus in the case of MANET, and a custom parser and AST available in the MATISSE framework. Our approach allows a weaver developer to use an already existing compiler, significantly reducing the effort when developing a weaver from the start. Finally, *Weaver Engine* refers to code that is actually implemented by the weaver developer. This is a fairly small part of the total code and illustrates how much work one saves by using our framework when developing a weaver for a new target language. In contrast, the amount of code of the LARA Framework is indicative of the possible effort required to start an AOP approach from scratch. It is a medium-sized project and we estimate that, with its current features and field-testing, represents well over an year of investment for a small team. Please note that the code of the LARA framework is shared by all three presented weavers. The compiler, interpreter and weaver generator used by these weavers is the same.

All the weavers in Figure 6 have been in development for at least a year and support rich language specifications. MANET is the weaver that has had most investment, and this is reflected in the amount of lines of code. Even though, their size is still a fraction of the LARA Framework, and in the case of MATISSE, it is an order of magnitude lower.

4.2 Techniques Impact

Table 1 presents several metrics taken from a set of LARA aspects that were developed following two approaches, one that uses APIs and actions provided by the weaver, and another that does not (API and Native in column Version, respectively). The former uses specific actions and LARA libraries (developed and distributed by weaver developers) providing functionality that intends to reduce user effort, native code insertion and accidental fault injection, while the latter only uses code insertion for the target concern. All aspects target C code and are compatible with the MANET weaver, except for the aspect `Func. Interface`, which targets Java code and is compatible with Kadabra.

The first metric, *LOC*, is the number of lines of code of the LARA aspect, while *NLOC* is the number of lines of native code (C or Java, in this case) that appear in `insert` actions. The column *Applies* represents the number of `apply` blocks. The column *Actions (Inserts)* shows how many weaver actions are executed and, from those, how many are the `insert` action. After that, *calls* represents the number of calls to other aspects (using the `call` keyword). The final two columns of the table show the reduction of the lines of aspect code and native code inserted (by the user) when moving from the native approach to the API approach.

These metrics represent static information about the aspects, as if they were written by an aspect-developer, and do not take into account the code inside libraries. Also, please consider that the number of lines of native code in insert actions in an aspect may not be representative of the number of lines of code that are actually inserted, since the same `insert` action can be apply on several points of interest.

The aspect *OMP* adds OpenMP pragmas to two loops. The native version inserts the necessary *include* and manually selects the loops and inserts the pragmas. The API version only has two calls, one per loop, which accept the name of a function and the name of a iteration variable to identify the loop, and the pragma to insert. In this case, it significantly reduces the number of lines of code.

The aspect *Type Def.* changes the data type of the variables in a given function. The native version textually replaces the original declarations with declarations that use the new type. The API version uses the action `def` to redefine the attribute `type` of the declaration, performing type validation and symbol table update.

The aspect *DCG* injects code that enables the generation of a dynamic call graph when the program executes. The aspect inserts counters for each $< caller, callee >$ pair and increments them when calls occur. The native version uses raw code insertions to declare global variables in multiple files as well as defining helper functions to deal with the counters. With the API version, we manage to greatly reduce the amount of inserted native code by using a code generation library. We also use an action to declare global variables, which lessens the amount of work needed and provides more safety.

The aspect *Func. Interface* creates a Java functional interface based on a given method, and replaces its calls with a field access. This aspect is important in the context of Java weaving, since it allows to safely insert arbitrary Java methods and fields outside of the scope of methods. The native code approach injects the new interface before the target method, introduces a new field of the interface type and replaces the call with an access to the new field. The API approach provides actions for interface extraction, field creation, and method call replacement, which significantly reduces the native LOC and makes the aspect safer.

The aspect *Time Calls* inserts code to time calls to a given function by wrapping calls with code to start and pause a timer. The aspect also inserts code to print the result at the end of `main` function. Overall, the code insertion in the native version includes header directives in several files, global variables and timing code. The API version uses a library that automatically creates the timer and deals with its setup. The library exposes an API that provides the required code to control the timer, eliminating the need for the user to write native code.

Finally, *VariableRangeMonitor* is an aspect that injects code to monitor the ranges of a set of variables of a given function.

Table 1: Code metrics regarding the example LARA aspects implemented using two different approaches.

Aspect	Version	LOC	NLOC	Applies	Actions (Inserts)	Calls	LOC Red.	NLOC Red.
OMP	Native	23	3	4	3 (3)	0	78%	33%
	API	5	2	0	0 (0)	3		
Type Def.	Native	11	2	1	2 (2)	0	9%	100%
	API	10	0	1	1 (0)	0		
DCG	Native	34	7	3	6 (6)	0	12%	86%
	API	30	1	2	1 (1)	3		
Func. Interface	Native	34	5	3	3 (3)	1	56%	80%
	API	15	1	2	4 (0)	0		
Time Calls	Native	34	9	5	7 (7)	0	26%	100%
	API	25	0	3	4 (4)	1		
Var. Range	Native	65	23	3	4 (4)	1	-2%	57%
	API	66	10	4	5 (2)	4		

The native version inserts arrays to store the minimum and maximum values taken by each variable, as well as code to update the array and print the result at the end. The API version reduces the native code by half, but overall slightly increases the size of the aspect. However, since the new aspect uses a library for code generation and variable declaration, it provides more secure native code generation.

In aspects such as *Time Calls*, *DCG*, *Func. Interface* and *VariableRangeMonitor*, we replace the raw insertion of certain elements (e.g., global variables, methods, include directives) with actions and libraries that add or declare these elements in a more reliable fashion. With this, the weaver is responsible to verify the action, generate the necessary code and update its information (e.g., symbol table).

Aspects where the API approach leads to a significant reduction of native code represent cases were we can define the aspect in a higher abstraction level. Instead of relying on the direct insertion of code, we specify what functionality we intend (e.g., insertion of timers). In the case of aspects *OMP*, *Type Def.* and *Func. Interface*, we could replace all `insert` actions with other more specific actions and library calls.

The native code left in the API versions represent blocks of code for which our current techniques would be more cumbersome to use than native code insertion, and would not bring any apparent benefit. In these cases, we considered that it was better to continue to use the native code.

4.3 Target-Language Independent Aspects

Figure 7 uses the previously presented techniques to rewrite the aspect of Figure 1 in a target-language independent way. Line 1 imports the Logger aspect, which is instantiated in line 4. Lines 8-9 call an API function that prints a string before the calls in the given source code. Although this version is not that different from the original code, it has other benefits. On one hand, it allows the aspect developer to program in a higher abstraction level. On the other hand, it can be safer to use, since APIs can hide essential complexity (e.g., includes in C) and limit the insertion of native code by the aspect developer.

```
1  import lara.inst.Logger;
2
3  aspectdef LogCall
4   call logger : NewLogger();
5
6   select function.call end
7   apply
8    logger.log($call, 'before',
9       $function.name+'->'+$call.name);
10  end
11 end
```

Figure 7: An example of a target-language independent LARA aspect that is equivalent to the aspect specified in Figure 1.

We present this case as a motivational example of the opportunities this approach can enable. We do not think aspect developers should initially aim to write target-language independent strategies, as they should not sacrifice expressiveness and legibility for this purpose. They should, however, follow a *coding – refactoring – library* cycle, as with any other programming language. If it is possible to standardize a set of common join points, actions and library APIs, we think this can enable further reuse, more specifically if performed at the weaver developer level. Keep in mind that we do not think necessary for all weavers to implement all common features; given the breath of variety in programming languages, we consider partial (or none) compatibility between weavers to be perfectly acceptable.

5. RELATED WORK

Most AOP approaches extend the target language with AOP concepts. AspectJ [13] extends Java and aims at providing better modularity for Java programs. AspectJ describes pointcuts lexically (e.g., `call(set*(..))`) and has a very mature tool support[5]. AspectJ join points are limited to

[5]Spring framework (https://spring.io) and Eclipse plugin (https://eclipse.org/aspectj)

object-oriented concepts, such as classes, method calls and fields, and several works try to complement AspectJ.

AspectC++ [21] is an AOP extension to the C++ programming language inspired by AspectJ, and uses similar concepts, adapted to C++. Both AspectJ and AspectC++ do not consider join points related to local variables, statements, loops, and conditional constructs. AspectMatlab [1] is another example of an AspectJ-inspired language, for MATLAB in this case. It adds some distinctive features related with MATLAB programs, such as the ability to capture multidimensional array accesses and loops.

Rajan and Sullivan [19] propose Eos-T, a version of the aspect-oriented language Eos. Both Eos and Eos-T extend C# with AOP concepts to include branches and loops as join points. Eos preceded Ptolemy [18], which uses an approach based on events instead of pointcuts. Ptolemy is focused on object-oriented languages and augments the target language so that it can define event interfaces and handlers. They provide a compiler that translates Ptolemy augmented-Java to plain Java. The triggering of events is inserted manually in the code where they should happen. They conclude their event-based approach is more resilient to changes than lexical pointcut descriptions, but is at a disadvantage when one intends to capture certain kinds of general pointcuts (e.g., all methods of a function) [8].

LARA has been inspired by many AOP approaches, including AspectJ and AspectC++, but differs from these efforts in several ways. Unlike most approaches, LARA has been designed so that it is decoupled from a specific target language. In a similar way, Jackson and Clarke [11] envision a language-independent approach, SourceWeave.NET, using an XML AOP language in the context of the .NET framework. This approach is tied to .NET and every new language one wants to support needs a parser that builds a CodeDOM graph, the representation expected by SourceWeaver.NET. Having the same intermediate representation allows the usage of reusable aspects, agnostic to the target language. However, the join points that one can select are already defined, coarse-grained and cannot be changed. These characteristics impose a limitation on what can be exposed from the target language and captured within aspects.

Other approaches address language-independent tools [15] for weaving existing components with aspects written in the language of choice. Others, such as UniAspect [16] target application components in different languages and use a common representation of the components to apply the aspects. UniAspect keeps a similar syntax to AspectJ and introduces `"@"` annotations for identifying the target language.

Compared to other approaches, LARA provides more flexibility in the join point model, which is based on composable select expressions (similar to functional queries [9]). Hence, LARA supports arbitrarily complex join point hierarchies, including different models of join points, e.g., MATISSE includes annotation-based join points using a proof of concept implementation based on comments. The way LARA supports attributes and actions is conceptually similar to the variables *thisJoinPoint* and *tjp* used in AspectJ and AspectC++, respectively, which contain meta-information related to the join point. However, attribute information in LARA is specified in the language description (attribute model) and can be extended by the weaver developer without changing any part of the LARA language.

There are a number of approaches that address concerns that are usually out of scope of traditional AOP (e.g., code transformations, compiler optimizations). CHiLL [20] is a declarative language focused on recipes for loop transformations. CHiLL recipes are scripts, written in separate files, which contain a sequence of transformations to be applied in the code during a compilation step. The PATUS framework [6] defines a DSL specifically geared toward stencil computations and allows programmers to define a compilation strategy for automated parallel code generation using both classic loop-level transformations (e.g., loop unrolling) and architecture-specific extensions (e.g., SSE). LARA takes a similar approach to source-to-source transformations with the use of actions, which are defined in the language specification and implemented by a weaver. One can select join points for optimization (e.g., loops), filter them based on their attributes and then apply transformation actions.

There are several term rewriting-inspired approaches for code analysis and transformation, such as Stratego/XT [3] and Rascal [14]. Such approaches require the complete grammar for each target language, which makes it possible to reuse the framework for different languages. Strategy reusability between languages is possible, as long as the grammars have common parts. LARA, on the other hand, promotes the usage of existing compiler frameworks (e.g. Cetus [7], Spoon [17]) for parsing, analysis and transformations, and its join point model does not require a one-to-one correspondence to the provided AST. Another distinct feature of LARA, for the weaver developer side, is that weavers can be built in an incremental fashion, adding join points, attributes and actions as needed.

6. CONCLUSIONS

This paper focused on the use of LARA in the context of multiple target languages and presented recent improvements to the LARA technology. We briefly presented LARA, an AOP approach that is independent of the target language, and extended this approach with techniques to overcome some of its challenges, more precisely, specifying additional behavior. We discussed the impact of the proposed techniques, and showed that this approach improves aspect code reuse, enables more concise and safer aspects and that the LARA framework can significantly reduce the effort needed to support new target languages.

As future work we intend to further explore the opportunities provided by a single-language approach to AOP in projects that require several target languages and customized tool flows. On a separate note, LARA currently performs static source-to-source transformations. We intend to extend the LARA framework to support dynamic actions, as a way to instruct the weaver that the LARA code in those actions should be executed during application runtime, instead of relying only on insertion of native code.

Acknowledgments

This work was partially funded by the ANTAREX project through the EU H2020 FET-HPC program under grant no. 671623. Tiago Carvalho acknowledges the support provided by Fundação para a Ciência e a Tecnologia, Portugal, under Ph.D. grant SFRH/BD/90507/2012.

References

[1] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. AspectMatlab: An Aspect-oriented Scientific Programming Language. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, New York, NY, USA, 2010. ACM.

[2] D. August, K. Pingali, D. Chiou, R. Sendag, J. Y. Joshua, et al. Programming multicores: Do applications programmers need to write explicitly parallel programs? *IEEE Micro*, (3):19–33, 2010.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52 – 70, 2008.

[4] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov. Lara: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 179–190. ACM, 2012.

[5] J. M. P. Cardoso, J. G. F. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonçalves. Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Software: Practice and Experience*, 46(2):251–287, Feb. 2016.

[6] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[7] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, 2009.

[8] R. Dyer, H. Rajan, and Y. Cai. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 143–154. ACM, 2012.

[9] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *Asian Symposium on Programming Languages and Systems*, pages 366–381. Springer, 2004.

[10] J. Fabry, T. Dinkelaker, J. Noyé, and E. Tanter. A taxonomy of domain-specific aspect languages. *ACM Comput. Surv.*, 47(3):40:1–40:44, Feb. 2015.

[11] A. Jackson and S. Clarke. Sourceweave. net: Cross-language aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, pages 115–135. Springer, 2004.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, volume 1241, pages 220–242. Springer, 1997.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[14] P. Klint, T. Van Der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[15] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *ACM SIGPLAN Notices*, pages 1–12. ACM, 2003.

[16] A. Ohashi, K. Sakamoto, T. Kamiya, R. Humaira, S. Arai, H. Washizaki, and Y. Fukazawa. Uniaspect: a language-independent aspect-oriented programming framework. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, pages 39–44. ACM, 2012.

[17] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, pages 1155–1179, 2015.

[18] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *European Conference on Object-Oriented Programming*, pages 155–179. Springer, 2008.

[19] H. Rajan and K. Sullivan. Eos: Instance-level aspects for integrated system design. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 297–306, New York, NY, USA, 2003. ACM.

[20] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A programming language interface to describe transformations and code generation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 136–150. Springer, 2010.

[21] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: An aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.