Expressing and Applying C++ Code Transformations for the HDF5 API Through a DSL

Martin Golasowski[1], João Bispo[2], Jan Martinovič[1], Kateřina Slaninová[1] and João MP Cardoso[2]

[1] IT4Innovations National Supercomputing Centre,
VŠB - Technical University of Ostrava,
17. listopadu 15/2172, 708 33 Ostrava
Czech Republic
(martin.golasowski, jan.martinovic,
katerina.slaninova)@vsb.cz,
[2] Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias, s/n 4200-465,
Porto, Portugal
jbispo@fe.up.pt, jmpc@acm.org

Abstract. Hierarchical Data Format (HDF5) is a popular binary storage solution in high performance computing (HPC) and other scientific fields. It has bindings for many popular programming languages, including C++, which is widely used in the HPC field. Its C++ API requires mapping of the native C++ data types to types native to the HDF5 API. This task can be error prone, especially when working with complex data structures, which are usually stored using HDF5 compound data types. Due to the lack of a comprehensive reflection mechanism in C++, the mapping code for data manipulation has to be hand-written for each compound type separately. This approach is vulnerable to bugs and mistakes, which can be eliminated by using an automated code generation phase. In this paper we present an approach implemented in the LARA language and supported by the tool Clava, which allows us to automate the generation of the HDF5 data access code for complex data structures in C++.

Keywords: HDF5, Domain Specific Language, LARA, source-to-source, aspect oriented, Clava, code generation

1 Introduction

Source-to-source transformation is a process during which a program source code is automatically created or updated according to a given set of inputs. It can be used for various tasks, such as low-level optimization for a given target platform, templating, integration and more. In this paper, we demonstrate how we can overcome the lack of compile-time reflection in C++, by applying user-defined transformations written in an aspect-oriented domain-specific language.

Reflection is the ability of a computer program to examine and/or modify its own structure. It usually provides information about the type of a given object, its inheritance hierarchy, its attributes and more, and can even be used to manipulate the code itself during run-time. This ability is available in many interpreted languages, such as Python, Ruby, Lua, Java or C#, usually thanks to the underlying virtual machine or interpreter.

The current version of the C++ language is C++17. Its features have been recently added into commonly used compilers such as the GNU GCC or Intel C++ Compiler. However, reflection is not among these features yet, though several proposals has been recently published [8]. Based on the speed of implementation of the new standards in mainline compilers and their adoption by programmers, it can be said that C++ does not have comprehensive support for compile-time reflection yet. We briefly mention several alternative tools and approaches to this problem in Section 2.

One of the common use cases for reflection is mapping an object to a persistent data structure, where individual attributes of the object are examined and stored in a proper way. In the use case presented in this paper, we are storing a complex data structure representing a traffic navigation routing index in a HDF5 based binary file. Without proper reflection, we have to manually create the code that maps the C++ structures to the objects in the HDF5 file. This code implements several time-consuming data processing tasks that are executed on an HPC cluster, which places severe constraints on the robustness of the code and the entire process.

In this paper we present a method, based on the LARA language, for automatic generation of the mapping code. Section 3 explains the routing index and its HDF5-based storage. Section 4 presents the LARA language and its toolset, which can be used to define the desired code transformations in a robust and flexible way. Section 5 shows a concrete application of the approach on our data processing code and its integration in our build process.

2 Related Work

There are several approaches to reflection in the C++ language. One of them is through extensive use of macros to annotate individual classes and attributes, a solution that is popular for example among game engines [2]. Its pitfalls are the inability to use reflection on non-modifiable code (e.g., third-party libraries) and its reliance on uncommon language constructs. A similar approach can be implemented using templates, at the cost of an increase in complexity of the code, compilation times and requirements for its maintenance.

Another approach is based on external tools which parse source code and have a certain knowledge of the code structure, such as the Meta object compiler, which is part of the Qt GUI framework. This tool produces source code for annotated C++ classes extended with support for accessing run-time information and a dynamic property system [5, 13]. This tool, however, provides only a fixed feature set intended for development in the Qt framework.

Domain-specific languages (DSLs) such as LARA can provide the desired level of flexibility and robustness for our purposes. The LARA language has been inspired by AOP approaches, including AspectJ and AspectC++. AspectJ[10] extends Java in order to provide better modularity for Java programs, and has a very mature tool support. AspectJ join points are limited to object-oriented concepts, such as classes, method calls and fields, and several works try to complement AspectJ. AspectC++ [14] is an AOP extension to the C++ programming language inspired by AspectJ and uses similar concepts, adapted to C++.

In traditional AOP approaches, aspects usually define behavior which is executed during runtime, at the specified join points. LARA differs from traditional AOP in that it uses aspects to describe source code analysis and transformations, which currently are executed statically, at compile time. Due to this difference in approach, tools like AspectJ and AspectC++ usually do not consider join points which are common in LARA, such as local variables, statements, loops, and conditional constructs.

There are several term rewriting-inspired approaches for code analysis and transformation, such as Stratego/XT [6] and Rascal [11], which require the user to provide a complete grammar for the target language. On the other hand, LARA promotes the usage of existing compiler frameworks (e.g. Clang [1] in the case of this work) for parsing, analysis and transformations. Another distinct feature of LARA is that *weavers* can be built in an incremental fashion, adding *join points*, *attributes* and *actions* as needed (see Section 4).

3 Hierarchical Data Format for Routing Index

Binary formats offer efficient and fast data storage. However, custom implementations can be cumbersome and fragile, especially in multi-platform environment. The Hierarchical Data Format [9] (HDF) provides a binary storage format implementation for storing large volumes of complex data. It has been developed mainly for storing scientific data, however, since then it has been adopted by many other industries. The HDF allows easy and consistent sharing of binary data across various platforms and environments, which is one of its main advantages. There are two main versions of the HDF format. In this paper, we exclusively refer to the HDF5 version [9]. HDF5 implements a storage model which resembles a standard file system hierarchy, with a tree of folders and files. The basic HDF5 file objects are *Groups*, *Datasets* and *Attributes*. Groups can hold one or more datasets; both groups and datasets can have attributes associated. Each HDF file has one root group. The datasets are used for the actual storage of multi-dimensional data of a given type.

3.1 Routing Use Case

The HDF5 provides APIs for a large number of major programming languages such as Python, C/C++, Java or even CLI .NET. Our codebase is written mainly in C++, hence we refer to the native HDF5 C++ API in this paper.

In our approach for graph data for traffic navigation routing index, individual road segments, junctions and other elements of a road network are represented by a set of vertices and oriented edges. The edges have associated a number of parameters such as length, max. allowed speed or category. Graph representation of a road network of single country such as the Czech Republic can have millions of vertices and edges. The vertices and edges in the HDF5 file are divided in subsets (graph parts) which reside in their corresponding groups. Mapping of the graph parts to the individual vertices is stored in the NodeMap dataset located in the root group of the file. The parts can be determined either by geography or other topological properties of the graph. Each graph part group then contains the Edges, EdgeData and Nodes datasets. All datasets in our case are two-dimensional, where rows hold individual records and columns hold their attributes. References to records in other datasets in our case are represented by storing an index of the referenced record rather than using the native HDF5 reference mechanism.

The edges reference their metadata stored in the EdgeData dataset. There is only a limited number of unique values of the edge metadata, hence it is efficient to store them in a separate dataset. Relationship between nodes and edges is represented by the edgesIndex column in the Nodes dataset which references rows in the Edges dataset. The Edges then hold reference to the Nodes via their ID. Graphical visualisation of the routing index structure is in Figure 1.

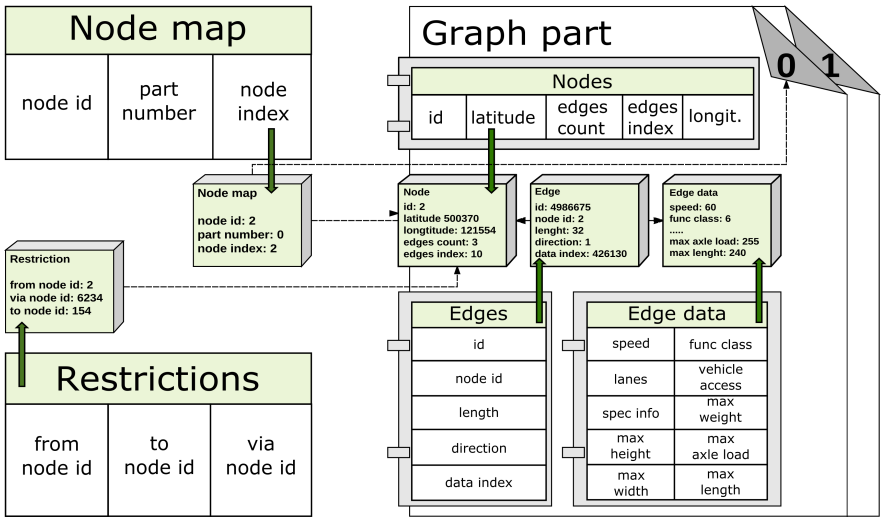

Fig. 1. Routing index layout in a HDF5 file

4 C++ Code Manipulation

LARA [7][4] is a Domain-Specific Language (DSL) for source-code manipulation and analysis, inspired by Aspect-Oriented programming (AOP). It has specific keywords and semantics to query and modify points of interest (i.e., *join points*) in the source code, and provides general-purpose computation by supporting arbitrary JavaScript code. Join points provide *attributes*, for querying information about that point in the code, and *actions*, which apply transformations to that point.

Figure 2 presents LARA code which adds `include` directives to a file, using a join point action. Line 1 declares an *aspect*, the top-level unit in LARA (which is similar to a function). Line 2 declares the inputs of the aspect, which in this case is a `file` join point. By convention, na mes of variables that represent join points are prefixed with a dollar-sign (`$`) in LARA. Line 4 uses a `select` to capture all the classes definitions that appear in the current program. Lines 5-8 represent an `apply` block that performs some work over the join points captured in the previous `select`. In this case, it executes a file *action* (`$targetFile.exec`) that adds an include directive to the file, corresponding to the file that belongs to the given join point (`addIncludeJp($class)`). This example shows a common pattern in LARA, which is to select some points in the source code and then act over them, possibly modifying the source code.

4.1 Clava

Unlike most source-to-source approaches, LARA was designed to be independent on the target language, which allows the LARA framework to be reused for several languages [12]. This was achieved by decoupling the specification of the points of interest from the LARA language. To use LARA code to transform a specific language (e.g., C++ in this case), we need to build a tool (called *weaver*) which connects the language specification to the target code representation, e.g., an Abstract Syntax Tree (AST).

```
1  aspectdef AddClassInclude
2   input $targetFile end
3
4   select class end
5   apply
6    // Add an include to $targetFile for the file where class is declarated
7    $targetFile.exec addIncludeJp($class);
8   end
9
10 end
```

Fig. 2. A simple LARA aspect that inserts, in a given file, an include directive for every class that appears in the source code.

Figure 3 shows Clava [3], a C/C++ weaver we developed that uses the LARA framework to enable C++ code manipulation[3]. Clava is mostly implemented in Java, and internally uses a binary based on Clang [1] to dump information about C/C++ programs. This information is then parsed and used to build a custom AST, which the weaver client uses in the queries, modifications and source-code generation specified in LARA code (which is interpreted by the LARA framework).

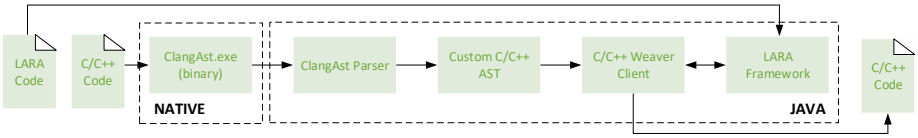

Fig. 3. Block diagram of the tool Clava.

```
#include <H5Cpp.h>

struct NodePosition {
    int nodeId;
    int partNumber;
    int nodeIndex;
};

class NodePositionType {
 public:
  static H5::CompType GetCompType() {
   H5::CompType itype(sizeof(Routing::NodePosition));
   itype.insertMember("nodeId",
    offsetof(Routing::NodePosition, nodeId), H5::PredType::NATIVE_INT32);
   itype.insertMember("partNumber",
    offsetof(Routing::NodePosition, partNumber), H5::PredType::NATIVE_INT32);
   itype.insertMember("nodeIndex",
    offsetof(Routing::NodePosition, nodeIndex), H5::PredType::NATIVE_INT32);
   return itype;
  }
};
```

Fig. 4. Generated compound type code for the `NodePosition` structure

5 Use Case

In this section we present and explain the LARA code developed to automatically generate type mapping functions from classes and structs (henceforth referred

[3] an online demo version is available at `http://specs.fe.up.pt/tools/clava`

to as *records*) present in the source code. The presented version generates a new class for each record found in the code, and this class has a single static method that returns a *CompType* object, which can then be passed to the HDF5 API calls when that particular record is accessed. In the example in Figure 4, for demonstration purposes, we include the code in the same file as the original record. However, in the code presented in this section we create new files for the generated code, to avoid adding a dependency to HDF5 in every source-file that wants to use the record (note that both cases can be expressed in LARA). Currently, the type-mapping code is generated for all classes and structures in the given source files, but the code can be easily adapted to filter unwanted records (e.g., by providing a list of class/struct names, or files).

5.1 LARA for HDF5

Figure 5 presents the *use* relationships for the aspect `Hdf5Types`, which generates HDF5 interface code for C++ records. It uses a LARA aspect, `RecordToHdf5`, which generates the implementation code for a single record, and two *code definitions*, a LARA mechanism for writing parameterizable escaped code (see Figure 7). The aspect `RecordToHdf5` uses a JavaScript function, `toHdf5`.

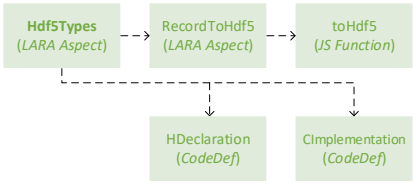

Fig. 5. *use* relationships for the aspect `Hdf5Types`.

Figure 6 shows the code for a working version of the `Hdf5Types` aspect. As input, it receives a path to the base destination folder of the generated code, and a namespace for the generated functions, with optional default values for the inputs (line 2).

Lines 5-6 use a Factory provided by Clava (i.e., `AstFactory`) that allows the creation of new AST nodes, that can then be inserted in the code tree. The `AstFactory` always returns join points, which can be handled the same way as the join points created by `select` statements. In this case, two join points of type `file` are created, one for the header file (*CompType.h*) and another for the implementation file (*CompType.cpp*).

Lines 8-11 select the `program` join point and add the newly created files with the action `addFile`. Line 15 selects all the records in the source code that are either of kind `class` or of kind `struct`, which are then iterated over in the `apply` block in lines 16-27. This block creates the declarations for the header

and the implementation file using the code definitions in Figure 7 (lines 20 and 25, respectively). It also adds to the implementation file an `include` directive for the current record (line 23), creates the code for the body of the implementation function by calling the aspect `RecordToHdf5` (line 24) and inserts the code of the function in the implementation file (line 26).

Lines 30-32 finish the header file by adding an include to the HDF5 CPP library, creating the namespace and inserting the code created in the `apply` block into the file. Lines 35-36 finish the implementation file by adding two necessary includes.

```
1  aspectdef Hdf5Types
2   input srcFolder = "./", namespace = "HDF5Types" end
3
4   var filepath = srcFolder + "/lara-generated"; // Folder for the generated files
5   var $compTypeC = AstFactory.file("CompType.cpp", filepath); // Create files for
6   var $compTypeH = AstFactory.file("CompType.h", filepath); // the generated code
7
8   select program end  // Add files to the program.
9   apply
10   $program.exec addFile($compTypeC); $program.exec addFile($compTypeH);
11  end
12
13  var hDeclarationsCode = "";
14
15  select file.record{kind === "class", kind === "struct"} end // Iterate over records
16  apply
17   var className = $record.name + "Type"; var typeName = "itype";
18
19   /* CompType.h file */
20   hDeclarationsCode += HDeclaration($file.name, className);
21
22   /* CompType.cpp file */
23   $compTypeC.exec addIncludeJp($record); // Add include to the record file
24   call result : RecordToHdf5($record, typeName); // C/C++ type to HDF5 type
25   var cxxFunction = CImplementation(namespace, className, code);
26   $compTypeC.exec insertAfter(AstFactory.declLiteral(cxxFunction));
27  end
28
29  /* CompType.h file */
30  $compTypeH.exec addInclude("H5Cpp.h", true); // Add include to HDF5 CPP library
31  hDeclarationsCode = 'namespace '+namespace +' {' + hDeclarationsCode + "}";
32  $compTypeH.exec insertAfter(AstFactory.declLiteral(hDeclarationsCode));
33
34  /* CompType.cpp file */
35  $compTypeC.exec addInclude("CompType.h", false); // Add includes for
36  $compTypeC.exec addInclude("H5CompType.h", true); // for CompTypes
37
38 end
```

Fig. 6. LARA code for the aspect `Hdf5Types`.

Figure 8 shows the code for the LARA aspect `RecordToHdf5`, called in the previous aspect. `RecordToHdf5` iterates over all the fields in the record given as input (line 7), ignores all fields that are constant (line 9) or not public (line 10) and creates the code for the specific type of the field using the JavaScript

```
codedef HDeclaration(filename, className) %{
//  [[filename]]
class [[className]] {
 public:
 static H5::CompType GetCompType();
};
}% end

codedef CImplementation(namespace, className, body) %{
H5::CompType [[namespace]]::[[className]]::GetCompType() {
[[body]]

    return itype;
}
}% end
```

Fig. 7. Code definitions `HDeclaration` and `CImplementation`, used in aspect `Hdf5Types`.

function `toHdf5`, ignoring cases that are not supported (lines 13-14). Features that appear for the first time in this example are the use of the attribute `type` (lines 5, 9 and 13), an attribute common to all join points in Clava and that returns a special kind of join point that represents a C/C++ type; and the use of inlined escaped code in the lines 15-17 (i.e., `%{...}%`)

```
aspectdef RecordToHdf5
 input $record, typeName end
 output code end

 var recordType = $record.type.code;
 code = "H5::CompType "+ typeName +"(sizeof("+recordType+"));\n";
 select $record.field end
 apply
  if($field.type.constant) continue; // Ignore constant fields
  if(!$field.isPublic) continue; // Ignore private and protected fields

  fieldName = $field.name;
  var HDF5Type = toHdf5($field.type);
  if(HDF5Type === undefined) continue; // Warning message omitted
  var offset = %{offsetof([[recordType]], [[fieldName]])}%;
  var params = %{"[[fieldName]]",[[offset]], [[HDF5Type]]}%;
  code += %{[[typeName]].insertMember([[params]]);}% + "\n";
 end
end
```

Fig. 8. LARA code for the aspect `RecordToHdf5`.

The Clang compiler has a very rich AST with detailed information, not only about the source code itself, but also about the types used in the code, which are also represented as an AST. Clava takes advantage of this information and

gives access to this AST for types by providing a join point `type`, which can be accessed from any join point using the attribute `type`.

The JavaScript function `toHdf5` (Figure 9) uses the attributes of the join point `type` extensively to generate the code for the HDF5 interface. The function starts by *desugaring* the type (line 8). Clang supports type *sugaring*, which means that if, for instance, we define in C/C++ a custom type `typedef int foo` and declare a variable `foo a`, `a` will appear in the AST as having the type `foo`, and not `int`. The attribute `desugar` returns the desugared version of the corresponding type (or the type itself, if it is already desugared).

Next, there are several special cases which need to be handled. For instance, C++ enumerations can customize the underlying integer type. If the type is an enumeration, the function is called recursively for the integer type of the enumeration (lines 10-12). Other example is the case of `vector` types, which appear in the AST as a `TemplateSpecializationType` (i.e., any type template that has been specialized, such has `vector<int>`). In this case, the function is also called recursively, this time for the specialization type.

After handling the special cases, the function uses the attribute `code` to obtain the code representation of the type and consult the table `HDF5Types`, which maps C/C++ types to the corresponding HDF5 types.

Table 1 shows several code metrics[4]. The code for the aspect totals 84 lines of code (LoC), including LARA code, Javascript code and code definitions, and generated around 100 lines of code for this use case (note that the aspect code is generic, and can be used for other use cases). If the generated code had to be written by hand, it would represent about 24 % of the LoC of this use case.

5.2 CMake Integration

Since Clava is a Java program, a Java runtime is the only system dependency required to execute the LARA aspect. Clava uses Clang underneath, and packages custom pre-compiled binaries for Windows, Ubuntu and CentOS platforms. The integration is done by defining a custom build step via *add_custom_command()* which produces the generated files and adds them as dependencies to the executable targets defined in CMakeLists. This integration allows a seamless use of this LARA toolset within a single build process.

Table 1. Code metrics for the use case.

Use Case				**LARA**	
#files	**#records**	**#fields**	**LoC**	**Aspects LoC**	**Generated LoC**
15	10	47	308	84	98

[4] LoC for LARA aspects where counted by hand. LoC for C++ code uses the L-SLOC value provided by LocMetrics (http://www.locmetrics.com/).

```
var HDF5Types = {}; // Table with mapping between C/C++ and HDF5 types
HDF5Types["int"] = "NATIVE_INT";
HDF5Types["float"] = "NATIVE_FLOAT";
HDF5Types["uint16_t"] = "NATIVE_UINT16";
... // Other mappings

function toHdf5($type) {
  $type = $type.desugar; // Desugar type

  if($type.kind === "EnumType") { // Special case: enum
   return toHdf5($type.integerType);
  }

  if($type.kind === "TemplateSpecializationType" && // Special case: vector
   $type.templateName === "vector") {

   var templateType = '&' + toHdf5($type.firstArgType);
   return 'H5::VarLenType('+templateType+')';
  }

  ... // Other special cases

  var HDF5Type = HDF5Types[$type.code];
  if(HDF5Type === undefined) return undefined; // Warning message omitted

  return 'H5::PredType::' + HDF5Type; // Base HDF5Type
}
```

Fig. 9. JavaScript code for the function `toHdf5`.

6 Conclusion

In this paper, we presented a possible solution to missing support for compile-time reflection in C++. Our solution is based on the domain-specific language LARA, which is used to write source-to-source transformations, and the tool Clava, which executes the LARA code over C/C++ programs. We have demonstrated its usage by generating a native C++ API for the HDF5 library, without modifications in the original source code. The generated code is used to store a traffic navigation routing index for processing on HPC infrastructure. Our use case is complex both in terms of structural complexity and data volume, and we needed to implement a robust and flexible approach to generate the data access code and integrate it into our build process. In Section 5.2 we introduced a basic approach for integration of the code generation process in CMake, by using custom build commands. The Clava tool is called during the build configuration to produce the type mapping code between C++ and HDF5 API.

Ongoing work includes adding support for custom compound types (e.g., fields that are user-defined classes/structs) and LARA and Clava support for custom `#pragma` constructs in the code, that can be used to mark arbitrary blocks of code to be processed by the LARA aspects. This approach can be used to apply a large number of custom optimizations (e.g., in the context of HPC systems) or to generate a concrete implementation of the data access layer on top of an existing abstract data storage library.

Acknowledgment

This work has been partially funded by ANTAREX, a project supported by the EU H2020 FET-HPC program under grant 671623, by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project 'IT4Innovations excellence in science - LQ1602' and co-financed by the internal grant agency of VŠB - Technical University of Ostrava, Czech Republic, under the project no. SP2017/177 'Optimization of machine learning algorithms for the HPC platform'.

References

1. Clang. `clang.llvm.org`. Accessed: 2017-02-28.
2. Unreal engine documentation. Online: https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Reference/index.html, 02 2017.
3. J. Bispo. Clava: C++ language + lara weaver and code transformer - antarex technical report v0.1. 2017.
4. J. Bispo and J. M. Cardoso. A matlab subset to c compiler targeting embedded systems. *Software: Practice and Experience*, 47(2):249–272, 2017.
5. J. Blanchette and M. Summerfield. *C++ GUI programming with Qt 4*. Prentice Hall Professional, 2006.
6. 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.
7. 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 *Procs. of the 11th annual int. conf. on AOP Soft. Dev.*, pages 179–190. ACM, 2012.
8. M. Chochlık. Implementing the factory pattern with the help of reflection. *Computing and Informatics*, 2015.
9. M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.
10. J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: aspect-oriented programming in Java*. John Wiley & Sons, 2003.
11. 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*, pages 168–177. IEEE, 2009.
12. P. Pinto, T. Carvalho, J. Bispo, and J. M. Cardoso. Lara as a language-independent aspect-oriented programming approach. In *Proceedings of the 32th Annual ACM Symposium on Applied Computing*. ACM, 2017. to appear.
13. Qt. Qt documentation. Online: http://doc.qt.io/qt-5/why-moc.html, 02 2017.
14. O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: An aspect-oriented extension to the c++ programming language. In *Procs. of the 14th Int. Conf. on Tools Pacific*, CRPIT '02, pages 53–60, Darlinghurst, Australia, 2002.