FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Expedient Logging for C++ Using Reflection

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Klaus Bräutigam
Matrikelnummer 0825965

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Wien, 21.04.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Expedient Logging for C++ Using Reflection

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Klaus Bräutigam
Registration Number 0825965

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Vienna, 21.04.2015

(Signature of Author) (Signature of Advisor)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Klaus Bräutigam
Ameisthal 44, 3701 Großweikersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

Many thanks to my family for assistance in all manner of situations, to my friends for their patience and lenience in busy times and my advisor for his helpful mentoring.

Abstract

From the very beginning of programming, the invisibility of software processing hindered effective interaction with human beings. Rising scale and complexity of increasingly parallelized and distributed applications demanded advanced programming languages and additionally progressive programming concepts like reflection. Logging and tracing to selectively reveal information from executed imperative programs with assistance of specialized tools, emerged early and is important till today.
The present work is concerned with the research question: *Is the development of practically usable logging tools with "automatic message scope enrichment" and "human-readable object serialization" possible by using existing approaches to equip C++ with reflective capabilities?*
"Automatic message scope enrichment" means that log messages are automatically equipped with extended information. Examples are thread ids during execution or method signatures to reveal the origin of the log statement in the source code. "Human-readable object serialization" deals with serialization of C++ objects with complex types (classes) during runtime, suitable for debugging tasks and software auditing. Next to pure feasibility of reflective logging for C++ also the introduced resource consumption overhead is focused on. Since C++ does not intrinsically provide sufficient reflective power, use of external frameworks is mandatory.
To solve the problem we start with general studies about reflective logging and required features in the context of C++ including related theory and publications. The next part deals with analysis and evaluation of available C++ reflection tools and outlines already existing reflective possibilities. A prototypical implementation is presented and the resulting characteristics are explained. Finally, the solution is evaluated concerning resource consumption and possible extensions for the future are suggested.
Evaluation results show that in general powerful reflection can be realised within the natively compiled C++ domain using available reflection tools. Also other mechanisms (e.g. persistence, dependency injection, etc.) beside logging can be realised with the described concepts. The solution reduces source code intrusion and uses reflection also for configuration of the logging tool. This approach supports better convenience for users. Resource consumption overhead for working memory is shown to be not really significant. On the one hand "automatic message scope enrichment" can be implemented with acceptable run-time performance overhead. On the other hand this kind of overhead is higher for "human-readable object serialization" limiting its practical use.

Kurzfassung

Seit den Anfängen der Programmierung erschwert die verborgene Ausführung von Programmen die Nachvollziehbarkeit durch den Menschen. Wachsende Größenordnungen und Komplexität von Applikationen, welche immer verteilter und nebenläufiger operieren, erfordern immer mächtigere und fortschrittlichere Programmiersprachen und Konzepte wie „Reflection". „Logging" bzw. „Tracing" entwickelte sich früh zu einem wichtigen Instrument, um mit Hilfe spezieller Werkzeuge relevante Informationen während der Programmausführung in für Menschen lesbarer Form zu extrahieren und hat bis heute große Bedeutung.
Diese Arbeit beschäftigt sich mit der Beantwortung der folgenden Forschungsfrage: *Ist es möglich, mit den heute verfügbaren „Reflection Tools" praktisch einsetzbare Werkzeuge für „Logging" in C++ zu entwickeln, welche Funktionen wie „Automatic Message Scope Enrichment" und „Human-Readable Object Serialization" unterstützen?* „Automatic Message Scope Enrichment" bedeutet, dass „Logging"-Nachrichten automatisch mit Informationen erweitert werden, welche deren Wirkungsbereich beschreiben. Dies kann beispielsweise während der Programmausführung der Identifikator eines „Threads" sein oder auch der Ursprung im Quellcode der Software. Neben dem Beweis der Machbarkeit der beschriebenen „Logging"-Mechanismen für C++ mit „Reflection" wird auch dem entstehenden Mehrverbrauch an Ressourcen große Bedeutung beigemessen. C++ unterstützt „Reflection" nicht in ausreichender Form, wodurch der Einsatz externer Frameworks unumgänglich ist.
Um das Problem dieser Arbeit zu lösen, beginnen wir mit allgemeinen Studien über „Reflection" und die „Logging"-Grundlagen und -Funktionen, inklusive relevanter wissenschaftlicher Publikationen. Danach beschäftigen wir uns mit der Analyse und Evaluierung verfügbarer C++ Werkzeuge für „Reflection". Darauf aufbauend wird ein lauffähiger Prototyp präsentiert und dessen wichtigste Charakteristika beschrieben. Zuletzt wird die Lösung bezüglich ihrer Laufzeiteigenschaften evaluiert. Vorschläge für weitere wissenschaftliche Arbeiten beenden diesen Teil.
Die Ergebnisse offenbaren, dass „Reflection" bereits in ausreichendem Maße von Werkzeugen für die nativ kompilierte C++ Domäne unterstützt wird. Die erörterten Konzepte erlauben auch die Umsetzung anderer Mechanismen (z.B.: „Persistence", „Dependendy Injection"). Die beschriebene Lösung reduziert die Quellcodedurchdringung und verwendet „Reflection" auch für die Konfiguration des Werkzeugverhaltens. Der zusätzliche Arbeitsspeicherverbrauch, bedingt durch den Einsatz von „Reflection", konnte als gering ermittelt werden. Die Funktion „Automatic Message Scope Enrichment" verursacht einen akzeptablen Overhead an Laufzeit. Die Ausführung von „Human-Readable Object Serialization" benötigt erheblich mehr Laufzeit und kann deshalb momentan nicht für zeitkritische Anwendungen eingesetzt werden.

Contents

1 Introduction **1**
1.1 General Information 1
1.2 Motivation 4
1.3 Problem Statement 5
1.4 Aim of the Work 9
1.5 Methodology 11

2 Analysis **15**
2.1 Genesis of Reflection 15
2.2 Adoption to Object-Oriented Programming 16
2.3 Adaptation for C++ 16
2.4 Aspect-Oriented Programming in C++ 20
2.5 Reflective Logging Features 21

3 Reflection in the Context of Logging **23**
3.1 Logging and Reflection 23
3.2 C++ Reflection Tool Evaluation 24
3.3 Reflection Tool Capabilities and Characteristics 29

4 Architecture and Configuration **33**
4.1 Build Intervention Process 33
4.2 Composition Architecture 36
4.3 Execution Space Architecture 42

5 Reflective Logging Implementation **45**
5.1 Tool Set 45
5.2 General Aspects 46
5.3 Automatic Message Scope Enrichment 49
5.4 Human-Readable Object Serialization 55

6 Evaluation and Future Work **61**
6.1 Functional Evaluation 61
6.2 Usability and Integration Evaluation 69
6.3 Open Issues and Limitations 70

6.4 Future Work 71

7 Conclusion **73**

Appendix **75**

Appendix A Transformative Benchmark Program 75

Appendix B Base Criteria Evaluation Results 78

Bibliography **85**

CHAPTER 1

Introduction

1.1 General Information

This section provides generic information about the document typography and structure. An accompanying example is introduced to support explanations throughout the work.

Typography and Document Elements

For the present work some basic rules are defined:

1. Footnotes are exclusively used for complementary information
2. Shorter code excerpts are depicted in dedicated figures and in

   ```
   console font style
   ```

3. Longer code excerpts and groups of figures or tables are collected in an appendix
4. In the following cases text phrases are emphasised using an *italic/cursive* font style:

- Whenever the meaning of a name is mainly restricted to the context of this work (e.g. feature names, program names, component names, function names, etc.)
- Names of used external tools
- Textual citations from other authors
- Programming language constructs (e.g. C++ types, short code snippets, operators, etc.)

Structure of the Work

- The introductory Chapter 1 motivates the general topic and explains the environment this work fits into. Secondly, it defines the problem being actually addressed and the research aim. The chapter continues with a description of the methodology. The procedural principles are presented for solving the problems declared previously for the present work.

- Chapter 2 summarizes publications about the topics reflection, AOP (aspect-oriented programming) and logging. They form the fundamental theoretic base for this thesis. We start with the roots of reflection and how it founds its way into OOP (object-oriented programming). Then the adoption to the C++ language is discussed for the different variants. The chapter finishes with the publications in the context of advanced logging. They provide the background for selecting the scientifically relevant reflective logging features, implemented within this thesis.

- Chapter 3 contains on the one hand general information about reflection and logging to support understanding of basic conditions and surrounding aspects. On the other hand the evaluation of available C++ reflection tools is described with two identified frameworks being examined in more detail. This evaluation encourages understanding of the tool architectures and implementation details.

- Chapter 4 presents at first an overview about the architecture of the prototyped reflective logging tool. It explains the input objects, processing steps, resulting output objects, contained components and the most important mechanisms.

- Chapter 5 enumerates the tool set used for this thesis. Afterwards it dives into the details concerning realization of the already described abstract concepts. It also partially deals with revealed consequences and resulting characteristics of the obtained solution.

- In Chapter 6 the implemented reflective logging tool is carefully analysed and verified. The evaluation is primarily concerned with the most important factor run-time performance. Aside from that, possibilities for scientific investigations in the future are outlined for the topic.

- Finally Chapter 7 summarizes all important deduced facts and points out the results of this thesis.

- The appendix provides auxiliary information for completeness reasons and supports deeper understanding of particular aspects.

Accompanying Example

To generate a relation to the real world and to support clearer explanations an accompanying example is introduced now. It is intended to complement the presented information throughout the work. The financial domain is chosen because of our corresponding knowledge and experiences in this area.
The global financial system provides a platform for trading various different kinds of assets approximately in real-time. The system is distributed among several continents and countries and coordinated by specific local institutions (e.g. banks, stock exchanges, etc.). Today trading decisions are increasingly accomplished by fully automatised systems through exhaustive computational market analysis. Under certain conditions human interaction must be possible to react to exceptional situations. Such trading systems must act in a globally distributed manner and cope with tremendous amounts of data. Their overall complexity and the necessity for quick updates of certain internal aspects rises. Growing trading frequencies enforce the use of efficient hardware, operating systems and performance maximizing software to be competitive. Every trade must be specified precisely and with correct timing. Otherwise high losses arise. Auditing and detailed documentation of all events, processes, calculations and decisions are a further cornerstone within profitable trading systems. All things considered the following considerations can be identified:

- Event-based distributed communication among the whole world
- High amounts of data
- Fast reaction times based on complex calculations
- Transactionally scoped processing of trades
- Detailed auditing and documentation enforced by law or enterprise internal policies

Tremendous amounts of money can possibly be earned with effectively built and serviced automatised financial trading systems. For this reason high interest can be observed in the area. Essential for successful systems is for sure sufficient domain knowledge about the financial markets. Nevertheless, suitable implementation of the applied software systems gets an increasingly crucial part of the whole picture.
C++ is a programming language widely accepted for performance and data processing intensive application domains. The presented characteristics of automatised trading systems form an ideal example for the research topic of the present work. As will be outlined the auditing and documentation requirements are directly addressed by efficient reflective logging in C++. Furthermore, reflection in general is also applicable to many other aspects like for example transactional safety and data processing. The example is generally named *Exemplary Trading Platform* and used in different circumstances in this work. It will be referred to with the abbreviation *EXTP*.

1.2 Motivation

Information processing and the way computation is accomplished in modern computers today goes far beyond what human brains can perceive on the fly. Programmers usually use logging mechanisms with suitable powerfulness to reveal human-readable information from their developed or maintained programs [39]. For example, the heavy use of basic console outputs for debugging pieces of source code shows the widespread relevance of logging for programming processes. Carefully positioned log messages allow software developers to obtain important and simplified information about what is actually happening during program execution. Generally it does not matter at which complexity grade programmers operate. Effective use of logging can be observed in small experimental applications and also within huge legacy software systems.

Logging in Changing Environments

Software systems reach striking dimensions and get more complicated every day. Computation tends to become highly distributed and parallelized. Our example *EXTP* from the financial sector proves this fact. Several subordinated trading nodes must be executed in different countries to support fast access to the local markets. Many different assets from different asset classes must be analysed in parallel and relations among them get enmeshed stronger. These circumstances additionally complicate the observability for humans. Existing well-tried tool-sets get progressively overstrained in certain situations. Conventional debuggers for step-wise execution and limited runtime variable introspection have been around for a long time. They are often taught to programming neophytes as the most effective possibility to reveal errors. It can be observed that these tools increasingly fail to successfully aid in solving difficult problems within the changing execution environments of modern programs. The application fields and tasks of software systems diversify and must be realized in shorter times and maintained by smaller groups of computer scientists. This situation heavily impacts the way such systems are designed, constructed, operated, maintained and replaced. Within the last years logging received growing attention. It emerged as a major aspect of software product quality in industrial and scientific areas [32]. Its excellences and advantages for improving development, debugging and maintenance processes themselves were discovered and started to be researched [78] [82]. The potential and possibilities provided by expedient logging mechanisms for more effective software development should not be underestimated. They should be seriously researched and revealed.

Necessity for Reflection

Expedient logging mechanisms and most other progressive software development tools usually rely on advanced concepts. These concepts like reflection are at best provided by the used programming languages and execution environments. They can be found in strongly different characteristics and also significantly vary concerning powerfulness. In the last decades virtualization received increasing attention to exploit advantages through abstraction from specific platforms. This circumstance can be observed on different layers like virtualized hardware resources, operating systems and also execution environments of programs. Many virtual machines are available accepting abstract machine code instead of non portable native forms. This circumstance was utilized to introduce already known advanced programming concepts with less effort. Although, reflection can be provided for native execution environments, it received more widespread importance within virtualized execution environments. An explanation can be that development in these abstracted directions basically shifts focus from resource consumption to other aspects. These aspects seem sufficiently important to be worth the acceptance of additionally introduced overhead. C++ source code is usually compiled and executed natively on related target platforms. This approach promises to be more resource preserving than others and is in many cases argued in that way. For this reason the language was designed and implemented with focus on other concepts but powerful reflection in mind. The fact that the language was designed with different concepts in the foreground does not mean that no others can be added. As long as no real antagonisms are identified with the language fundamentals, advanced development tool sets (e.g. reflective logging) can benefit from them. Appliance of reflection can also improve programming processes, source code and the overall resulting program quality. All things considered it becomes clear that researching expedient logging mechanisms in conjunction with reflection within C++ is worth to be paid adequate attention. This work is concerned with this topic.

1.3 Problem Statement

Although logging has not yet received such strong significance in research as other topics have, steady progress can be observed within the last years. In practice logging is widely treated as rather dispensable as opposed to being a major aspect of successful software engineering. Beside others one of the reasons seems to be that for many languages no really expedient logging tools exist. For the language C++ a number of different logging tools are available. They provide basic functionalities used to integrate rudimentary logging into developed software systems. To the best of our knowledge, many useful features are missing, leading to problems and inefficiencies. The artificial, abbreviated and simplified piece of C++ source code in Figure 1.1 serves as a pictorial representation of state-of-the-art logging often observed by us in several projects. Such logging code is often applied to industrial and scientific applications in similar forms using widespread logging tools. The code should provide insights into the way how some desired abstract logging features are realized and applied in practice. Several disadvantages and problems can arise from such approaches.

```
...
class ClassName {
    ...
    static LoggingLibrary::Logger classLogger;
    ...
    ClassName () {
        LOG_METHOD_DEF ( "::ClassName ()" );
        LOG_TRACE ( "Construction" );
        ...
        LOG_TRACE ( "Constructed" );
    }
    ...
    UInt32 methodName ( UInt32 argument1, char** argument2 ) {
        LOG_METHOD_DEF ( "::methodName" );
        LOG_DEBUG ( "ENTRY with " << argument1 ); // argument2 omitted
        ...
        LOG_INFO ( "Object State[M1: " << object.getM1 () <<
        "|M2: " << object.getM2 () <<
        "|M3: " << object.getM3 () <<
        ...
        "]" );
        ...
        LOG_DEBUG ( "EXIT with " << returnValue );
        return returnValue;
    }
    ...
};
...
LoggingLibrary::Logger ClassName::classLogger = LOG_LOGGER ( "ClassName" );
```

Figure 1.1: Simplified example for state-of-the-art C++ logging source code

Automatic Message Scope Enrichment

The first observable logging feature that is realized within Figure 1.1 is enrichment of log messages with certain scope information:

- class name
- method name
- thread identifiers
- logging of method entry and exit points

The intention behind is to reveal origins of log messages within the source code or from different threads and scope switches like method calls and returns. The form of this meta-information is in most cases completely identical for all source code parts and can be classified as boilerplate code. Because of the fact that such logging code is often hand-crafted, the consequence of frequent adaption mistakes during source code refactoring, movement or duplication arises. These mistakes continuously lead to confusion, unnecessary subsequent faults and tremendous waste of invaluable development time [81]. Such kind of meta-information should be added to log messages automatically, whenever requested by programmers with simple, adaptable and generalized mechanisms. This identified feature is within this work named *Automatic Message Scope Enrichment* and in the following text referred to as *AMSE*. To be implemented in a clean and generalised way it requires basic reflection capabilities. The scope of every log message statement must be deduced and added as meta-information. This information extension can be accomplished on the one hand at compile-time by automatic injection of scope information into the log message statement. On the other hand an executed log statement can reflect the information dynamically at runtime from a meta-facility. Both possibilities necessitate specific types of reflection categorized as compile-time and run-time reflection.

Human-Readable Object Serialization

The next popular logging feature concerns tracing of passed and returned variable values of subroutine calls and objects in general. Figure 1.1 shows that variable values are printed within methods to obtain additional contextual state information. Fully automatised and human-readable object serialization is a valuable mechanism. It is essential for powerful logging and has already been implemented for languages like Java [42]. For C++, if anything, only fundamental types can easily be serialized to human-readable formats [17]. All other more complex types must be manually equipped with suitable serialization procedures. Mechanisms like overloaded streaming operators are often used in such cases. This circumstance is unacceptable concerning development effort, maintenance and error-proneness and should be automatised as far as possible [38]. This feature is named within this work *Human-Readable Object Serialization* and in the following text referred to with the abbreviation *HROS*. As for *AMSE* basic introspective reflection is required for realization. *HROS* must deduce the member types and subsequently their names. Afterwards the serialized instance must be introspected, to extract the current values for all of these members. Additionally, the process must be considered to happen hierarchically and even recursively. An object can subsume other objects (also possibly itself) which maybe again subsume other objects and so on. Termination of the serialization procedure must be considered for data structures with cyclic references. The *HROS* feature can as *AMSE* be realized using different kinds of reflective mechanisms.
Many traditional logging tools would be even more worthwhile than today if they were equipped with such advanced functionalities. The features *AMSE* and *HROS* provide improved usability and benefits for users. They are not yet applied and the related drawbacks raise avoidable problems and inefficiencies in software development.

Tool Configuration and Adaptability

The way how functionalities can be accessed and suitably configured by users is an important aspect. Development tools should not only provide powerful functionalities, but also optimize the way how these are applied and adapted in different circumstances. Many features of widespread logging tools require boilerplate code and context specific hand-crafted extensions, limiting their

usefulness and effectivity.
To be really invaluable, a tool must consider its integration into the operation area. Most available logging tools force source code integration in an extremely intrusive way and are therefore hard to change, adapt, remove and control. A good example for intrusive logging code provides the source code example in Figure 1.1. It shows how work intensive changes like removing the whole logging code from the project would be.
In our opinion this waste of development time to appropriately configure and maybe adapt logging behaviour of software became a serious maintenance issue in many projects.

Reflection Support

The reason why most available tools lack features like *HROS* and *AMSE* is that they require support for advanced programming concepts like extensive reflective capabilities. Reflection in informatics means that a computer program is able to discover, manipulate and influence its own structure, configuration, properties and execution behaviour. Usually reflection should be provided through dedicated mechanisms lying within the programming language itself [60] [46]. Unfortunately, standard C++ provides only strongly limited [17] [10] and often problematic [45] built-in support for reflection. Due to the fact that it is important for some C++ applications [14] [49], extensive research has already been accomplished. In the past many tools, more or less powerful, were implemented and published with strongly diversified features [37] [55] [24] [16] [22]. They provide reflective extensions and functionalities for C++ from external. Many of these available reflective framework or library implementations for C++ were not designed under consideration of maximum general applicability. They were created more in the context of specific problem domains. For this reason in many cases, special mandatory environmental preconditions and influences can be identified. Unfortunately, this lack of general applicability strongly reduces reusability and therefore distribution to other application areas. Software projects tend to disassociate themselves from proprietary framework solutions to avoid problems. This circumstance seems to be one of the major reasons for the rare manifestation of advanced reflective concepts within the C++ domain. It should be resolved by successfully applying already available reflective frameworks and libraries to practically relevant and general problems. Clearly presented solutions can serve as examples to overcome common prejudices against such advanced concepts. Often they are just dismissed as superfluous and fancy ideas instead of being applied as serious improvements.

1.4 Aim of the Work

This section describes the aim of this work consisting of some subordinated aspects finally resulting in an explicitly formulated research question.

Reflection in General

A subordinated aim of the present work is to provide fundamental insights into the topic of reflection for the C++ language. It outlines different approaches researched in the past and their applicability and usability. Especially also usability, stability and functional range are evaluated for implementing advanced tools for software engineering in general.

Reflective Logging Features

Based on general analysis, the feasibility of implementing an advanced, portable and practically usable logging tool for standard C++ is to be presented. The implementation uses available reflective capabilities and a consecutive evaluation shows the reached practicability. A restricted set of two features is implemented using the best identified C++ reflection tools available. The two features *AMSE* and *HROS* are representative for proving basic feasibility.
The *AMSE* feature provides several mechanisms to automatically enrich log messages in programs with additional scope information. In this context the term scope is used to express circumscribable program structures or behaviour. Scopes can be classified for example by:

time Log messages from a procedure starting at timestamp x and ending at timestamp y are absolutely time scoped with two timestamps. In another example a procedure lasted 5 seconds, giving just a relative time duration scope of 5 seconds for log messages. Such time-based scope information is especially important for reactive processes.

declaration Log messages can relate to the same method, class or source code part. Important for object-oriented domains.

execution space Log messages origin from the same thread, process or operating system instance. Essential for multi-threaded and distributed applications.

execution behaviour Log messages origin from the same method call with a specific computation path.

priority Log messages have the same severity (e.g. ERROR, WARN, etc.) or semantic decoration (e.g. ALGORITHM, MODEL, etc.) attached. Suitable for better grouping and structuring of logs in a further dimension.

The following scope enrichment mechanisms are provided:

- class and method name a log message originates from
- thread id identifying the thread a log message comes from

- entry and exit logging for methods to provide scope among subsequent log messages
- time duration measurement of method calls

These mechanisms can assist software developers in several circumstances from the very beginning of projects until maintenance and replacement of the applications. Developers of our *EXTP* can efficiently locate programming mistakes within specific algorithms, asset tracking threads, trade timings or data models by applying suitably scoped logging. The generated logs can also be used for analysing the realized business processes. Of course all the outlined mechanisms already exist and are required by programmers. Different is in this case that they should be applicable automatically, consistently and easer than before using a reflective approach.
The *HROS* feature is concerned with serialization of instances with complex C++ types (i.e. *class* and *struct*) to human-readable text strings. For example, object states of assets, running transactions and fulfilled trades can directly be logged within *EXTP* and are automatically adapted when types are modified. The complexity of the C++ type system and historical influences provide big challenges. So one aim surrounding this feature is also to evaluate existing limitations that possibly prohibit a sound and complete solution. Such limitations can originate from characteristics of the C++ language or used external tools.
The two explained features *AMSE* and *HROS* solve some of the biggest drawbacks and problems of state-of-the-art logging. At least many open source projects suffer from lack of them [81].

Reflective Tool Configuration

Another important aspect of software engineering tools is concerned with configuration and adaptability. The intention behind is to use reflective capabilities also for interaction with the programmer, not only to implement functional features. Effective software development tools exploit all available possibilities to provide a maximum level of usability and powerfulness. Intrusiveness and forced prerequisites must be minimized. Showing possible solutions and implementations with assistance of reflection form another aim of this work.

Critical Evaluation

The announced logging tool is expected to serve as a general proof for applying reflective concepts within the domain of C++. Preconditions, restrictions and limitations are minimized. The tool must work with standard compliant C++ compilers on different platforms and avoids as far as possible the use of compiler dependent or non portable features. This approach encourages representativeness in general.
As already mentioned, beside the plain proof of implementation feasibility, this work aims on revealing the practical suitability concerning resource consumption. Especially for domains in which C++ is commonly used, resource consumption is an important point to be considered. Not only the general availability of useful features is of interest. At least as important, related costs are assumed to be presented with sufficient precision. For this reason the logging tool will be evaluated within two different application domains (reactive and transformative) [65]. Resource consumption (runtime and memory) is measured according to well-defined guidelines and the results presented in a clearly arranged manner.

Research Question

All things considered the aim of the present work is to answer the following research question: *Is the development of practically usable logging tools with AMSE and HROS possible by using existing approaches to equip C++ with reflective capabilities?*

1.5 Methodology

This section provides information about the applied methods and concepts of this work starting with the origin of the topic. This work, in general, systematically applies scientific methods to deduce the expected results.

Literature Study and Analysis

A very important scientific method is looking at related information and results already published by other scientists. For this reason the pure analysis and research of suitable concepts must be enriched with a systematical and extensive literature study. It is necessary to extract fundamental knowledge about the already researched and commonly known theory of reflection in general. Afterwards a special focus is given to the topic within the context of the C++ language. Obtained information in summarized form can outline the most important factors and proper categorization gives a clearly arranged overview. An important part of the literature study is to obtain knowledge and understanding about logging in software systems and the overall relevance for software development. It reveals the most significant features, identified by scientific work in software engineering in the past. These features form the base for research within the reflective logging work and outline concrete requirements for the applied reflective mechanisms. Literature can be read with different perspectives in mind which determine the focus given to specific details presented within the texts. In the context of this work we concentrate on three related, but distinguishable perspectives:

- At first fundamental insights into the concept of reflection as a whole and within the domain of C++ will be extracted.
- The second perspective deals with possibilities to apply this fundamental theory within tools for improving development processes.
- The third one, especially relates to literature dealing with the topic of logging. Focus is explicitly tied to the core intention of this work namely to acquire theoretic knowledge to successfully and effectively apply reflection to logging tools for the language C++.

Reflection Tool Evaluation

In analogy to the literature study, for practical scientific research topics also already published implementations must be taken into consideration. The practical part of the present work is based on use of currently available reflection tools for C++. They must be studied and analysed according to predefined methods and reasonable characteristics.

- Available tools which offer applicative reflective capabilities for the standard C++ language must be identified. Suitability is not always obvious at the first glance since reflective capabilities are often hidden between or within other superordinated features. For this reason the exploration of reflection tools for C++ must also operate aside the core terminology of reflection.

- After identification of potentially suitable reflection tools, they must be evaluated for their general applicability within the context of this work. Quality, stability and compatibility of the provided functionalities are crucial. Each one's adequacy is separately evaluated. With this approach the completely unusable subset is discarded from the entire set of reflection tools by verifying formal characteristics and properties.

- The remaining tools are functionally evaluated again in separation with lightweight prototypical implementations to prove pure feasibility of required reflective logging features in advance.

- The third evaluation part assumes that all remaining reflection tools are suitable on their own. Since it is unlikely that one reflection tool already provides all necessary features for the task, it is important to consider that several reflection tools must coexist. So they also must support interaction with each other. For this reason the best suitable reflection tools must be combined in another prototypical implementation to prove their ability to fulfil all mandatory requirements also in conjunction.

The presented strategy is composed of a theoretical part ensuring certain formal characteristics and properties to hold in advance of a time consuming in depth evaluation. A second rigorous practice-driven part directly targets and reveals the practical suitability of the remaining tools. The method applies the well-known divide and conquer paradigm. A preparatory in depth evaluation of subjects in isolation is followed by a subsequent evaluation in combination.

Logging Tool Implementation

To show or refuse the feasibility of implementing *AMSE* and *HROS* is one of the core tasks of this work. It is accomplished with a prototypical logging framework implementation using the reflection tools, resulting from the related evaluation. Important to mention is, that the focus is not tied to maximized development-friendly usability or finalized maturity of the logging tool. Priority lies more on the scientific aim of proving or disproving the general feasibility.
The architecture of the logging tool is based on the prototypical implementation which proves the used reflection tools to be compatible to each other. This decision is wise because the evaluation discloses all prerequisites enforced by the reflection tools. It is seldom the case that tools are designed with such vision and generic brilliancy that users can apply them on almost all of their own architectures and layouts. For this reason the necessities for effective reflection tool use are deduced directly from the evaluation results (prototypical implementations). This approach implies unpleasant restrictions that must be taken into account for the architecture (e.g. type meta-information provided in form of compiled C++ binaries).
On the foundations of the resulting architecture *AMSE* and *HROS* are implemented in an iterative

way. They are not implemented one after another, but iteratively in parallel. Starting with fundamental functionality for both and related test cases in advance the two features are alternately extended. So comparable maturity can be reached and none of both can occupy significantly more effort than the other one. This approach reduces the risk that in the end not all research aims can be reached because too much time was wasted on failing parts.
Also the reflective configuration is declared as a research aim and is tightly woven with both features. For this reason also for this task an iterative and alternating approach is chosen.

Logging Tool Evaluation

For *AMSE* and *HROS* resource consumption overhead must be measured and depicted. A set of test cases provides information about handling various detailed functionalities correctly (e.g. stack tracing, C++ exceptions, enums, etc.). The evaluation is expected to reveal, whether the researched approach fits practical requirements. The following parameters with their measured quantitative values are compared using special benchmark programs:

- runtime
- maximum working memory consumption
- application binary size

A limit of 50 per cent is defined as the maximum acceptable resource consumption fraction within the benchmark programs. If reflective extensions consume as much resources as the underlying mechanisms themselves do, the approach should be declared to be practically not applicable and refused. Pure analysis during development represents the exception, but in the end reflective logging must be completely removed from production variants.
The measured time and memory consumption overhead is just considered for the reflective logging tasks. The basic effort for executing logging statements, console output or writing to files on persistent storage, etc. is not seen as reflective overhead. Also available traditional logging tools require this overhead when writing static log messages to files.
Separate benchmark programs, representing different application domains, are implemented in this work. They are equipped with logging statements and built with different configurations:

1. No Logging - any log statement is reduced to its neutral representation before compilation
2. Static Logging - logging to a file on the local file system (no reflective computations, only log messages with predetermined static strings written)
3. Reflective Logging - logging with reflective mechanisms to a file on the local file-system

The three presented configurations provide direct comparability for different aspects:

- The cost of logging overall can be retrieved by simple subtraction comparison of measurement results from configuration one and two.
- The proportion of reflective effort can be calculated by configuration two and three.
- The last possibility is to reveal the complete costs of reflective logging against using no logging with configuration one and three.

In the end, careful examination and representation of measurement results reveals whether the scientific aim has been reached or not.

CHAPTER 2

Analysis

This chapter provides a comprehensive aggregation concerning evolution of reflection in C++ and corresponding publications. The topic of reflection in informatics is a superordinate concept and was originally not discovered and researched in C++. For this reason at the beginning also non-C++-related publications are enclosed. In the following also the topics AOP and reflective logging features are analysed.

2.1 Genesis of Reflection

In the literature the genesis of procedural reflection is in most cases seen in the work of Smith [60]. Although his thesis finished in 1982 and showed reflection in action for the programming language LISP-3, the original idea started in 1976 at the company Xerox. Smith worked on a program (MANTIQ) basically reasoning about itself concerning structure and behaviour. In his opinion the idea already existed before *Citation: "...procedural reflection is not a radically new idea; tentative steps in this direction have been taken in many areas of current practice. The present contribution - fully in the traditional spirit of rational reconstruction - is merely one of making explicit what we all already knew."* [60]. In his work, definitions and general properties of reflection and related calculi are given. In chapter 5 an example for applying reflective capabilities in computer programs is described. It deals with string-based debug output in functions, giving access to the computational process whenever a problem occurs during execution. This example can be seen as a basic ancestor of the ideas behind the *AMSE* feature.
In 1983 Batali published about the topic and provides an answer to the question what introspection is [12]. Four properties are described which must hold for any system with introspective behaviour. It is mentioned that reflection processing does only make sense for 2-levels. This fact means that thinking about the way to think about the way to think about something, does not really make sense. Batali claimed that strong relation and focus to human related concerns, like deliberation and perception, were drawn at that time and that this tendency decreased in recent scientific publications.

2.2 Adoption to Object-Oriented Programming

Maes [46] in 1987 described an experiment to adopt computational reflection for OOP languages. He provided a further definition of computational reflection and claimed that in previous publications mysterious orientation dominated in contrast to technical relevancy. A bunch of examples were propagated, outlining the substantial practical value of reflection (e.g. performance statistics, debugging information, step-wise execution and tracing, object serialization, etc.). Many of the mentioned ideas like tracing of function entries are related to this thesis.
The importance of differentiating structural and computational reflection was stated more precisely in 1989 by Ferber [29]. The two reflection tools used within this thesis can more or less be categorized to cover these distinct forms of reflection. *Reflex* provides more capabilities in the direction of structural reflection and *AspectC++* focuses more on the computational part. Ferber also explained the widespread and important term reification in the context of message sending processes among classes (i.e. method calls). In this case a common base class for all reflected classes and special message objects is suggested to solve the problem of computational reflection.

2.3 Adaptation for C++

One of the earliest publications for adoption of reflective concepts in C++ was tackled by Ping et al. [52] in 1990. Structural and computational (although here named operational) reflection were explicitly distinguished and separately dealt with. The difference between interpretation-based and compilation-based languages was outlined. This differentiation raised the question, whether reflective capabilities should be directly integrated into the compiler or added through external mechanisms. The solution was chosen to be meta-object based, similar to the approach described by Maes without compiler intervention. For this reason structural reflection on the one hand was conceptually implemented with three forced rules, all reflected classes must keep the properties:

- a single root class for all classes must exist
- the structure describing meta-object class of any class must be defined as subclass of it including a default constructor
- only virtual member functions are allowed to be used such that vtable (virtual function table) exploitation is possible

On the other hand computational reflection is accomplished with method diversion, which means that vtables are exchanged dynamically. The problem with this approach is, that it forces a specific inheritance hierarchy for the target application source code base. Hand-crafted meta-object class definitions using four specifically provided macros must in addition be maintained by the programmer. Tracing is mentioned to be one major target for applying the won reflective possibilities. A later approach by Stephens [64] in 2003 also forces inheritance from a specific root class to implement reflective capabilities.

In the same year Interrante et al. [35] pointed out that C++ provides no run-time information about class names, inheritance hierarchies and so on. They explained that manual and incompatible class extending mechanisms are no suitable substitute. They intended to create a standard (*"the dossier"class*) including a tool. This tool should generate necessary additional source code and add a data member to each defined class to support access to run-time type information. Generally the same major problems like in other reflection extensions remain. User defined classes must still be extended with specific properties and the use of virtual method definitions stays mandatory.
In 1992 the definition of reification was updated by Madany et al. [44] and extended with explanations about which attributes should be reified and why:

Storage Concurrency and performance for memory allocation and deallocation

Existence Mechanisms for automating the deletion of objects

Persistence Mechanisms for persistent object activation and deactivation

Class Relationship between object and its class

Inheritance Relationships between classes in a class hierarchy

Encapsulation Hardware-enforced encapsulation of instance data

Inspection Human-readable representation of objects for tracing and instrumentation

The availability of structural meta-information as a necessary part is stated, beside the objects address and object identity. This statement is crucial for designing and implementing the feature *HROS* of the present work. The authors explicitly do not advocate modification of the C++ language to establish support for the discussed reflective mechanisms. Such intervention would fundamentally change the language. They wanted to provide information about the topic from an operating systems perspective for designers founding new languages based on C++. Most of their ideas contribute also to other contexts.

Reflection with Preprocessing and Code Generation

Chuang et al. [18] provided in 1998 another approach for reflection in C++. At first the term introspection is defined to be a limited form of reflection. It just deals with analysis and modification of object states at run-time using generalized mechanisms. The term introspection became commonly used like that. Many frameworks existing at that time, like for example IBM SOM and Microsoft COM provided C++ object introspection by forcing inheritance from special reflection base classes. The newly presented framework provides introspection through parsing class declarations to create a support environment. If all class declarations of reflected classes are available at development time, meta-classes can be built either manually during development time or even automatically at build-time using a code generator. The resulting introspective

meta-information for reflected classes is not necessarily statically linked with the application binaries. It can be used loosely-coupled as dynamically linked library and loaded on demand only if necessary at run-time. As problems they mentioned limitations through ambiguities within the language C++, for example class data members of type *void** and unions and unavailable source code of third-party libraries. In relation to this thesis, the automatised generation of overloaded streaming operators « and » for serialization should be pointed out. It is claimed, that with the presented system, the original class declarations are not modified and that all class properties stay compatible in any way. Nevertheless, the mandatory insertion of special friend method declarations must be seen as limited, but not fully negligible source code intrusion. At some points even the static type system of C++ is bypassed, decreasing confidence in stability.
Chuang et al. [17] [19] also published a command line based tool providing automatised object serialization code generation through source code analysis. A detailed elucidation is given about problems with intrusive and non-intrusive concepts and platform independence (e.g. object memory layouts, etc.). The strong limitations concerning RTTI (run-time type identification) of C++ are pointed out, too. The examined approach is, like most others, also source code intrusive. The mandatory private member access forces insertion of friend methods into classes. The type safety may be problematic since pointers of type *void** are used for passing objects. Some more issues and details for designing reflection tools for C++ are given by Kasbekar et al. [37]. They introduced an analyser and code generator for transparent object persistence to databases.
In the year 2003 within the SEAL project a work using *GCC-XML* [40] for C++ source code analysis and *java.lang.reflection* [51] in mind for implementation was published by Roiser [56]. A comprehensive overview about many other approaches is provided and the limitations of them explained in relation to the restricted reflective feature set of C++. In 2005 Roiser et al. [55] updated the previous publication and officially presented the *Reflex* library of the SEAL reflection system. In the same year the *ROOT* data analysis framework was discussed [14] in relation to reflection and integrated *Reflex*. Later *Reflex* was completely migrated to *ROOT* [67] to form a complete framework for extensive data analysis, heavily used at the LHC (Large Hadron Collider at Cern, Switzerland).
Meanwhile in 2004 Madina et al. [45] provided additional insights into RTTI and presented another intrusive approach for introspective reflection in C++. They also applied macros and friend method insertion to access private class members. A further approach *reflcpp* followed 2007 by Devadithya et al. [24] providing in addition extensive information about source code intrusiveness and an interesting definition *Citation: "Intrusiveness refers to the requirement of having to add certain annotations to a class so that it is able to provide information about itself."*. This definition is of special interest since *Reflex* provides annotations, disguised as standard C++ source code comments discussed in more detail later in Section 4.2. This fact raises the question how to assess annotations concerning source code intrusiveness. Maybe the definition should be extended to include the impact of added annotations to the source code. The publication further mentions that *Reflex* uses offsets from object pointers to access data member values. This approach may be problematic for certain applications because it is not guaranteed by the C++ Standard [6] for non-POD types.
In 2008 Naumann et al. [49] summarized all previous approaches to deduce reflective meta-information (dictionary information) from C++ source code depicted in Table 2.1.

Approach	**Description**
Patched Compiler	meta-information deduced by compiler or comparable mechanism
Custom Parser	a special non compiler related parser is used for source code analysis and meta-information generation
Debug Symbols	compiler integrated debug information of binaries is exploited for reflective capabilities
Manually Maintained	programmers must create and manually maintain type related reflective meta-information

Table 2.1: General reflective information deduction approaches by Naumann et al. [49]

The efficiency of meta-information representation is discussed because extension of applications with reflective data increases resulting binaries. It is mentioned that using *Reflex* doubles the size of finally linked binaries and even more overhead is introduced by heavy appliance of templates. The authors claim that using another C++ parser *clang LLVM* [53] could reduce build time and memory consumption. The improvement is possible by omitting the necessity of generating, compiling and linking additional C++ source for reflective meta-information. Information from pre-compiled headers can be used instead. In this context also the publication from 2010 by Dos Reis et al. [26] is of interest since they describe IPR, a systematic representation of C++ as fully-typed abstract syntax tree. The intention behind was to create an infrastructure for convenient handling of C++ structures in several kind of programs like compilers for example. But this circumstance maybe also contributes to external tools dealing with reflective meta-information generation.
In 2012 a further framework *Mirror*, providing compile-time and run-time reflective functionalities for C++, was presented by Chochlík et al. [16]. It is a command line utility (*MAuReEn*) supporting automatised use for a variety of enumerated applications. Also another C++ parser *OpenC++* is mentioned with the drawback, that it is only working with older compiler versions from GCC (GNU Compiler Collection [31]) and therefore not suitable for future extensions. The framework also applies meta programming strategies including new features from C++11.

Reflection with C++ Templates

Based on the introduction of templates, some of the later approaches to equip C++ with reflective extensions tried to exploit the possibilities available at compile-time. One of the earliest adoptions was done by Attardi et al. [11] using templates and macros for generating meta classes of types for abstracted connections to relational databases. In the same year Attardi et al. [10] again dealt with the topic to overcome the limited RTTI functionalities of C++ through extensive use of C++ templates. They claimed that using templates instead of macros for definition of reflective meta-information results in better support through multiple compilers and platforms. Later in 2004, Zolyomi et al. [83] touched the topic not in the context of run-time reflection, but with valuable information about checking template definition correctness at compile-time with standard C++ language features.

In the year 2008, Kenyon et al. [38] announced CHIMP, a reflection tool using a two step imperative meta programming technique. Here templates were used for inspecting source code for structures, classes and class members. Also a code generator was included for generating introspective meta-information (referred to as dictionary information) non-intrusively. The biggest drawback of this approach is the high complexity through different tools and an additional script language embedded into C++.
Another approach using newly available RTTI features from C++11 was published in 2012 by de Bayser et al. [22]. They provide a comprehensive introduction into the possibilities arising from the use of *dynamic_cast*, *typeid* operator and variadic templates. The limitations for reflectively iterating class members (i.e. data members and methods) and deducing their qualifiers and values are still present as for all previous pure TMP (template meta programming) approaches.
In 2014 the reflection group of the C++ Standardization Committee SG7 published a proposal [59] dealing with extraction of members and types from classes using variadic templates.

2.4 Aspect-Oriented Programming in C++

According to our studies the first publications dealing with AOP-related topics in the context of C++ were published in 1999 by Willink et al. [77] [76]. They mention FOG (flexible object generator), a meta-compiler for weaving of multiple definitions, such that resulting C++ code satisfies ODR (one definition rule). A source-to-source translator converts super-set C++ source files to traditional C++ header and implementation files. This approach provides the power to define aspects separately for classes by introducing other classes and functionalities into them. These extensions for existing classes are subsequently combined using FOG to cover ODR.
In the year 2002 C++ was enriched by Spinczyk [62] with even more powerful computational reflection utilities. They presented *AspectC++*, a non-intrusive, aspect-oriented framework using PUMA as source code transformation system. A related scientific publication from 2011 by Urban et al. [71] further explains cross-cutting concerns, pointcuts, join points, code weaving and aspect-oriented basics in the context of C++. *AspectC++* is based on the idea and therefore strongly related to AspectJ [30] for the Java programming language, in all its facets. For the reason that the C++ grammar is much harder to analyse and fundamentally different than that of Java, some adaptions in the syntax for aspect and advice definitions were necessary. For example the use of the % sign instead of * as wildcard, since * in the C++ grammar represents pointer types. For scope definitions the delimiter :: is used to express C++ namespace nesting instead of . for Java packages. Apart from that, most other concepts could be transferred to C++.
In 2005, Yao et al. [79] announced another aspect-oriented framework named AOP++. It followed a different, source code intrusive approach based on TMP concepts. The obvious drawback is that hand-crafted TMP source code must be added to the target application. This additional effort leads to reservations among programmers and therefore decreases interest for practical implementations. The source code of this framework seems to be not freely available from any official source.
Although the offered possibilities through AOP frameworks sound auspicious, they never really received widespread approval within the C++ domain. The following publications show that *AspectC++* can be effectively applied to problems in specific application areas.

At first Mahrenholz et al. [47] dealt in 2002 with a monitoring tool for instrumentation and used *AspectC++* for analysis code weaving. The approach included time measurement similar to the *AMSE* feature of this thesis. It is claimed that *AspectC++* is much faster concerning execution than AspectJ. It is argued that the weaved source code is compiled into the application, providing optimizations like code in-lining. No cumbersome run-time reflection mechanisms are mandatory. All things considered, the publication is strongly related to the work of Spinczyk et al. [62] but gives some additional valuable information. Calafiura et al. [70] used *AspectC++* in 2004 for extensions of a HEP (high energy physics) framework for improved logging and some other applications.
In this context another publication from 2002 by Thomas [69] outlines relations between reflection and AOP. It is claimed that computational reflection forms the roots of AOP and that practical relevance of reflective and aspect-oriented programming concepts is underestimated.
Finally it should be mentioned that in fact more AOP frameworks for C (e.g. *AspectC*, *AspeCt-oriented C*, *Aspicere*, etc.) and C++ (*XWeaver* [33], *FeatureC++*, etc.) exist, but they, according to our studies, never received real attention in the scientific area and are therefore not covered in more detailed.

2.5 Reflective Logging Features

A significant number of scientific publications mention advanced logging and tracing as example of applying reflection and AOP. The available logging frameworks and features are just one side of suitable software application logging. In many cases the power already provided by existing logging tools is not sufficiently utilized. Suitable positioning and prioritization strategies for log statements are crucial for effective logging. Correct configuration and use of further advanced logging features are at least as important as the powerfulness of the logging tool itself is. A goal of the present thesis is to identify potential logging use cases requiring reflective capabilities. Yuan et al. [80] [81] identified and described major issues with log statements in analysed software applications. In addition they provided some guidelines to improve logging message design in general:

- Recording of thread ids within multi-threading environments
- Use *__FILE__* and *__LINE__* to extract the position of log messages in source code files
- Use of severities to support logging prioritization
- Record relevant variable values in logs

CHAPTER 3

Reflection in the Context of Logging

This chapter unifies the topics logging and reflection in the context of C++ and reveals reflective capabilities already provided by existing tools for C++. This is accomplished in form of a strategic exploration and evaluation of frameworks.

3.1 Logging and Reflection

Logging started to become more important for software systems and development in many different areas [32] [81] [80]. Some information sources for successful logging concepts exist and some companies apply systematic approaches to reach high quality and tremendous improvements [20] for their logs. On the other side many projects still suffer from fundamental misusage and mismanagement of logging mechanisms [23]. This circumstance is a bit surprising, since a lot of widespread modern languages (e.g. Java, C#, Pyhton, etc.) offer powerful reflective mechanisms. So they also support uncomplicated implementation of advanced logging features. Some related features are directly integrated and therefore naturally used. A good example in Java is the method ”toString()” to serialize an object in addition to a textual logging message. Without being properly overwritten, the method only prints the memory address of an object, but the Java Reflection API provides the power to generate this method automatically [42]. Integration of such features directly into the language design opens the gates for applying them also in other areas.

Language Internal Reflection for C++

Many successful and widespread languages are built around abstracted environments, for example abstract machines or source code interpreters. C++ is generally compiled to native, platform-specific formats and mostly used with the argument of being very resource preserving. The focus on this aspect becomes obvious when examining the ideas and purposes behind the development of the language [27]. Adding reflective capabilities was discussed [72] [59] in the C++ community and also rudimentarily realised with RTTI [22]. Limited support can be further seen in the introduction of template meta-programming [7] [10] [11] [83]. In our opinion a consistent integration of reflection has never been officially accomplished by the C++ committee. For this reason the standard C++ language does not really provide a convenient base for the development of advanced tools. They usually require powerful reflective mechanisms. The covered topic of expedient logging is one possibility for using reflection to automatise and ease certain tasks of software engineering.

Reflective Extensions for C++

Today users apply C++ for several reasons, but at the same time heavily depend on extensive reflection for their daily business. Next to our example *EXTP*, scientists at the LHC [49] or generally scientists working in the field of HEP (High Energy Physics) [14] in the past needed reflection support within C++. They started investigations in this direction to bridge the gap. From these groups in science and industry, tools were developed and published that promise to provide a certain amount of reflective functionalities for C++. Another topic strongly related to behavioural reflection came up with growing distribution of the aspect-oriented programming paradigm. Some languages applied this paradigm and provided powerful tools. *AspectC++* offers a broad variety of useful features for reflectively injecting code extensions at build-time into applications.

Reflective Logging for C++

To the best of our knowledge no one ever before directly focused on researching the applicability of reflection to expedient logging in the context of C++. Only basic approaches to veer towards such ideas exist. One well known logging library, boost.log [57], applies the standard C++ immanent function-local predefined variable *__func__*. It reveals scope information with the macro *BOOST_LOG_FUNCTION* for functions, methods and classes [58]. The C++ standard declares the actual form of the variable content as implementation-defined [6]. This declaration prevents portable use of the feature across different compilers. However, the library supports the use of this limited feature whenever suitable. Another library from Arne Adams [8] partially deals with the topic by offering hand-crafted macro definition-based reflective tracing. This approach seems to be practically not applicable in most cases due to its tremendous manual maintenance effort in addition to the restricted feature set it provides. Most logging tools provide different limited scope information about runtime execution, for example thread identifiers. Also special macros for marking specific scopes with proprietary identifiers are offered. Such identifier strings are automatically added to log messages until the end of the scope [58].

3.2 C++ Reflection Tool Evaluation

Strategy

The evaluation strategy focuses on two different groups of criteria. The following set of formal base criteria is evaluated on a theoretic level to eliminate tools which do not even fulfil the absolutely necessary fundamental requirements. The presented order also expresses prioritization:

- The powerfulness provided by reflection tools is prioritized high to maximize possibilities for implementation.

- Reflection for C++ in general and the prototyped logging tool of this work are intended to be as portable as possible. For this reason the portability criterion is also very important to support a broad range of application fields.

- The next important criterion is documentation since the best tool does not provide real power without information about how to exploit it.

- Maintenance is the only guarantee that tools are not only valuable at the actual time but also in the future. The overall age of a tool is also of interest in this context since emerging technology is in many cases not stable and mature enough to be of real value.

- The criterion independence means within this evaluation that as few third party tools, external libraries, etc. as possible are necessary to be used. Taking care of this aspect can reduce overall complexity of integration and use. Also the intrusiveness concerning source code is covered by this criterion. The perfect reflection tool should also as little as possible depend on the actual source code base it is applied to.

- The overall maturity of tools targets the community interaction, current users, communication with the development team, etc. and is categorized just with three levels (experimental, prototype and mature).

- The intention behind development tool gives an impression on the targeted application fields. Deduction of the feature set and the way these can be accessed is supported.

- The final formal criterion evaluated for the reflection tools is the license since the prototype framework must omit any problematic economical dependencies.

Criteria are evaluated obtaining specific characteristics depicted in Table 3.1.

Criterion	**Characteristics**
Powerfulness	number of features, diversity of reflective capabilities
Portability	restrictions for specific platforms/environments
Documentation	number of words, completeness for features
Independence	source intrusiveness, tool or language feature dependencies
Maintenance	update frequency UF/last update LU/Age
Maturity	experimental/prototype/mature
Intention	intentions behind design and targeted application fields
License	Open-Source/Free/Commercial

Table 3.1: Characteristics for evaluation criteria

Evaluation Subject Exploration

The set of evaluation subjects consists of the reflection tools identified during literature study which are mostly of scientific origin. Some tools were found through an exhaustive exploration on the internet using a powerful search engine [34]. The following C++ reflection tools form the set for base criteria evaluation:

- QT's meta-object compiler and system [43]
- ROOT Reflex [66]
- cpgf [54]
- Arne Adams - A Reflection Library [8]
- XCppRefl [25]
- CAMP [68]
- Classdesc [63]
- CppReflection [41]
- clReflect [75]
- idevkit [4]
- rttr [48]
- meta [73]
- autoreflect [13]
- crd [15]
- xrtti [36]
- Reflect [28]
- AspectC++ (aspect-oriented) [62]
- XWeaver (aspect-oriented) [33]

Base Criteria Evaluation

In this evaluation phase, available information from all libraries is carefully collected and analysed according to the criteria from the defined strategy. The detailed evaluation results for every reflection tool can be referred to in Appendix B.
From the base criteria evaluation we conclude that only two reflection tools are adequate and worth to be evaluated in a practical manner. Most of them are problematic concerning the criterion ”powerfulness” and suffer from insufficient documentation. One of the major drawbacks is the independence criterion. The predominant fraction of tools does not provide real automatised reflective capabilities, but relies on macro-based or other hand-crafted meta-information. This information must be provided by the application programmers in addition. This nuisance is clearly defined to be avoided in the present work. Some promising tools must also be rejected because they are explicitly implemented for a specific operating system or compiler. Because of the outlined deficiencies only the following two tools seem to be possibly suitable and are evaluated in more detail:

- *Reflex*
- *AspectC++*

Functional Evaluation in Separation

During separated functional evaluation, for both tools the necessary sources and binaries are obtained from their related portals and prepared in dedicated development projects. With the obtained knowledge from examples and tutorials dedicated project setups are created to understand the functionalities provided by the frameworks.
For *Reflex* the following introspective capabilities seemed most valuable and dedicated experimental code was written and executed in the related evaluation project.

- Obtaining type information data structure (*Reflex::Type*) from a class instance (*typeid*) or by type name
- Extracting meta information (e.g.type name, member methods, namespace nesting, etc.) for a specific type
- Iteration of types with stored introspective meta-information
- Deducing of annotation types added to a type or method, including handling of related annotation attributes
- Construction of instances from type meta information
- Handling and conversion of fundamental data types within the *Reflex* type system
- Iteration and structural analysis of data members for a type
- Obtaining data member values for instances of types

- Introspective modification of object states (i.e. selective changing of data member values)

AspectC++ needed a project setup with an aspect header file (*.ah) and sources for code weaving. The following functionalities have been explicitly verified in the *AspectC++* evaluation project:

- Entry and exit points of the main function were equipped with weaved in logging code
- Several pointcut definitions were used to verify the expression matching patterns
- Construction and destruction advices were applied to weave in code at object creation and destruction time
- The differences of call and execution advices were analysed; template class code weaving needs call advices; conventional class code weaving needs execution advices
- Obtaining meta information from join point (e.g. method signature, unique identifier, etc.)
- Use of slicing to weave in data members and methods into existing classes
- Accessing data member values of objects by type information

The evaluation of the two frameworks is based on the described tests and experiments with their provided functionalities. Their suitability for powerful reflection in C++ could be proved.

Functional Evaluation in Combination

Combination of both reflective tools is done using the *Reflex* evaluation project and step by step extending it with functionality from *AspectC++*. The integration of the code weaving mechanism reveals a major issue. Introspective meta-information source files generated from *Reflex* cannot be processed with *ag++* from *AspectC++*. The sources include non-supported C++ code leading to the mandatory separation of reflective processing at compile-time. Chapter 4 will be dedicated to explain this process in detail. To give a short preview the major aspect of the necessary separation is that introspective reflective analysis with *Reflex* must be done before code weaving. The introspective meta information is at first extracted from the source code base and compiled to related binaries. After the code weaving process and compilation of the resulting source, the binaries can be linked together.
The obtained perception is used to successfully combine the two frameworks in a single project and to exploit their collective reflective powerfulness. The following points are important results from this evaluation phase:

- Using introspective capabilities of *Reflex* in weaved source code parts
- Use join point meta information obtained from *AspectC++* in *Reflex* mechanisms
- Separate the build process of the application into several phases to exploit full power of annotation-based configuration

- Share information globally between logging components and weaved source code parts

The evaluation of both reflective tools in combination revealed further immanent characteristics of them. These characteristics imply certain limitations and restrictions for the prototyped reflective logging tool. On the other hand also the reflective possibilities could be verified from the practical perspective.

3.3 Reflection Tool Capabilities and Characteristics

In this section we outline the characteristics of both reflection tools with their strengths and weaknesses.

ROOT Reflex

The *Reflex* reflection tool was founded within a project called *SEAL* [55] and in the end of 2005 was adapted and integrated into *ROOT* [67] (data analysis framework used at LHC). It can be found in the literature under different names. The tool was intended to analyse existing source code without the necessity for any markings or extensions added. It is fed directly with source files. As a consequence separate meta-information source files are generated from them. The generated meta-information source files can afterwards be independently compiled and linked for any purpose. This approach guarantees complete non-intrusiveness and supports at the same time full automatisation of the reflection process at build-time.
Reflex is based on *GCC-XML* [40], a powerful C++ language parser using the C++ front-end to GCC for creating a complete representation of C++ source code in XML. Although *GCC-XML* is of unpayable value and an important step in progress towards more improved C++ development tools, its limitations should not be ignored. They arise from the circumstance that such analysis mechanisms are not directly enforced to be embedded within the language by the standard. Deviations can occur concerning which language features are supported by *gcc* and *GCC-XML*. Source code parsing may yields different results, although the tools are closely related to each other. As an example, *GCC-XML* cannot deal with atomic operations necessary within *gcc* to approximately support the C++11 memory model [3]. This issue prevents the use of *GCC-XML* within any source code applying the widespread boost::thread [5] library at that time. For this reason such restrictions arising from the background must always be taken into account and dealt with effectively.
Nevertheless, *Reflex* provides a bunch of valuable reflective features which can serve for an incalculable number of purposes within advanced programs. Features are mostly related to the C++ type system and therefore of structural nature. *Reflex* deduces meta-information from existing source code and builds a compiled and run-time-accessible table of types from it. This table is organized using the compiler's internal unique type identifiers returned also by the C++ language operator *typeid*. Reflective information about the use and internal structure of known types is provided. It can be obtained at run-time using the rich API established within the *Reflex* namespace named Reflex::. The most important reflective capabilities of *Reflex* are on the one hand that the complete naming of types with enclosing namespaces is possible. From arbitrary

reflected types, most names of data-members, nested types and also inherited types can be discovered. Even new entities (methods, data members, etc.) can be added whenever necessary. On the other hand the complete structural composition of types can be ascertained, including type meta-information of all data-members, nested types and inherited types again. In addition to the structure, types with related names of data-members and inherited types, also the exact memory position of values in instances can be obtained. This functionality supports nearly complete introspection of objects at run-time. All these capabilities enable unimagined possibilities for any problem that concerns types in general, their structures and state of instances. The powerfulness in this specific area of reflection is strongly related to the foundational reasons leading to the development of this reflection tool. *Reflex* is used for automatised high performance object persistence tasks operating on huge amounts of data [55]. The described capabilities and characteristics are also viable for automatised reflective logging of object states.
Another feature provided by *Reflex* are annotations which can be added to C++ classes, structs and class methods. They can be accessed at run-time to express meta-information for several different use cases. Annotations are well-known in other programming languages like Java [50] and similarly as attributes in C# [21]. What is special about annotations from *Reflex* is that they appear in form of standard C++ comments. For this reason, they are guaranteed to have no influence on conventional interpretation during C++ preprocessing and compilation. Placed annotations only have meaning in the context of reflective analysis with *Reflex*. They provide useful possibilities for configurational tasks and at the same time do not introduce target source code intrusion affecting run-time behaviour.

AspectC++

AspectC++ is a software development tool supporting AOP features for the C++ language. It is strongly related to *AspectJ* [30] for the Java programming language. A number of scientific publications are referenced in Chapter 2. Sources and binaries are freely available on the internet. In general, aspects with contained advices can be defined in so called aspect header files with the default file extension *.ah. In the context of our example *EXTP*, possible applications of the available AOP mechanisms are outlined in Figure 3.1.
Advice definitions are supported in different sorts (call, execution, constructor or destructor) and must be chosen correctly. For example, methods of template classes must be declared as call pointcuts. For non-template classes execution pointcuts are mandatory to work appropriately. Pointcut expressions support template parameters and hierarchical namespaces and are therefore suitable also for non trivial tasks. For intercepting method calls three different types of advices are provided (before, after and around). They support behaviour enrichment separately before the method call and after return. Also around-invoke definitions for both cases in a single advice definition are possible. Order-advices can be defined to prioritize competing advice definitions. In addition to behavioural extensions for method calls also the structural constitution of types can be changed. New data members, methods and inherited superclasses with slice advices can be added to existing classes.
AspectC++ provides access to certain meta-information for an actual join point. Examples are a unique identifier, function or method signatures and line numbers in the related source code file. Furthermore, the types of data members, class names and pointers to data values for specific

instances can be obtained. These functionalities enable introspective reflection of object states at run-time. The only drawback is that they are designed as template methods with the argument numbers being specified as template parameters. For this reason, the structure of the type cannot be dynamically iterated as it is possible with *Reflex*. So the type structure must be completely known at advice generation time. This requirement complicates implementation of introspective mechanisms.
The practical use of *AspectC++* is based on two different GCC related executables called *ag++* and *ac++*. They are fed with aspect header files (*.ah) in conjunction with source code files. The defined code extensions are then weaved in. Two different processing results can be requested:

- Source files containing the weaved in extensions as C++ source code
- Compiled binaries of the source code base containing the weaved in functionalities already in form of machine code

```
// EXTP aspect header example
// An interceptor forced by law to intercept trades on the financial market
aspect LawfulTradeInterceptor {

    // match all trade entities in the source code base
    pointcut TradeEntities() = "extp::trade::%";
    // match all execution methods of trade entities with void return parameter
    pointcut TradeExecutions() = "void extp::trade::...::execute(...)";
    ...

    // generic class with specific law enforced properties
    slice class LawfulTradeMarker {
        unsigned long recordIdentifier;
        ...
        public:
        void getRecordIdentifier () {
            return recordIdentifier;
        }
        ...
    };

    // Let all matched trade entities inherit from LawfulTradeMarker
    advice TradeEntities () : slice LawfulTradeMarker;

    // all trade executions must be notified to a controlling trade inspection
    advice execution ( TradeExecutions () ) : around () {
        TradeInspection::getInstance ().notifyTradeInception ( this.toString () );
        tjp->proceed(); // actual call of the void execution(...) method
        TradeInspection::getInstance ().notifyTradeCompletion ( this.toString () );
    }
    ...
};
```

Figure 3.1: Example for applying AOP to *EXTP*

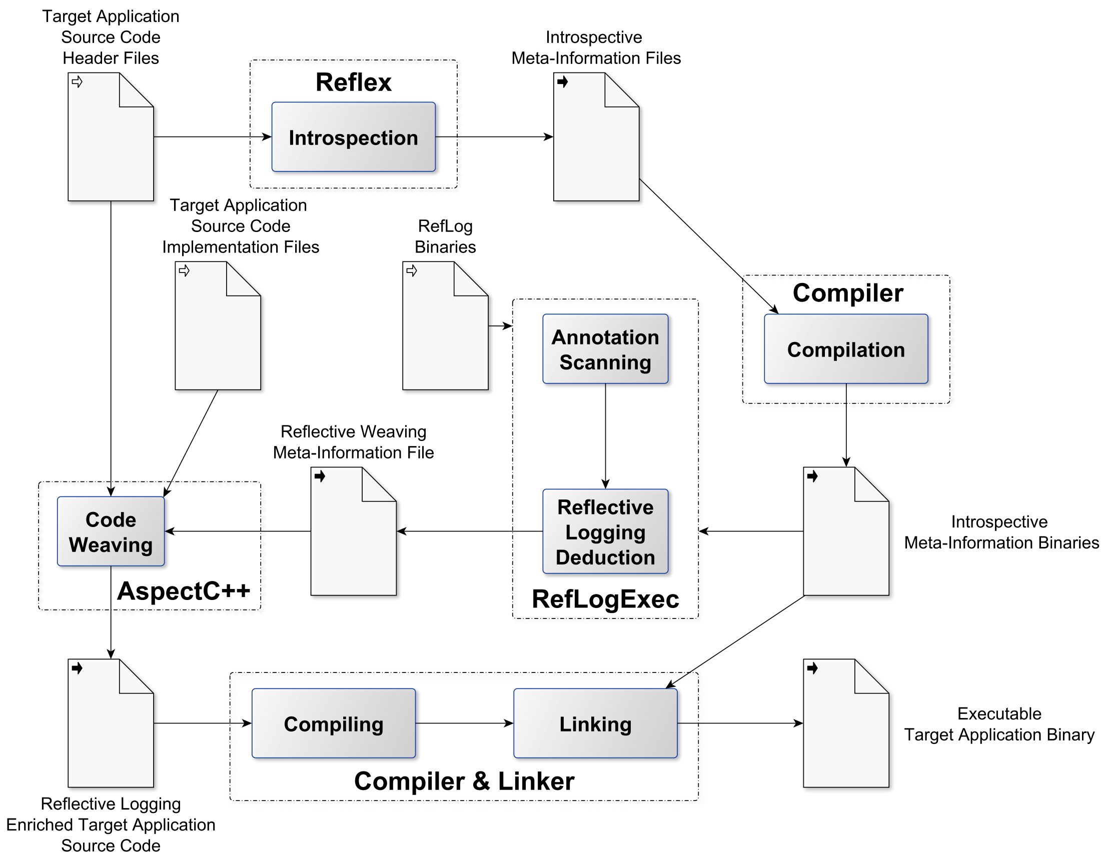
CHAPTER 4

Architecture and Configuration

This chapter describes the architecture of the prototypical logging tool implementation in the following referred to as *RefLog*. At first an explanation about, how the building process of an application that makes use of *RefLog* must be intervened. Then the internals of the *RefLogExec* application are presented. The last part deals with how *RefLog* is embedded into the source code of applying applications.

4.1 Build Intervention Process

This section explains necessary modifications for the build process of an arbitrary application using *RefLog*. In the following it is referred to as target application. Figure 4.1 presents this build intervention process in graphic form to provide a clean overview at a glance. The intervention is mandatory to support sufficiently powerful reflective capabilities within C++ applications with the identified reflection tools. Usually a program build just includes the compilation of the source code base and linking of related compiled binaries. The introduced expanded process incorporates certain activities and components to enable and entirely exploit reflection tool functionalities. Their purposes and procedures are described in the following to provide a solid background about the architecture of the reflective logging tool. Also the appearing input objects, intermediate objects and output objects (source files, binaries, etc.) are explained. To really understand the process described here, it is indispensable to distinguish and internalize the different levels and dimensions of reflection. *RefLog* applies reflective concepts for introspection of types at build-time to extract meta-information necessary for the target application build process itself. The same meta-information is used in addition later at run-time for the *HROS* feature. Another form of build-time reflection is used to configure and inject logging code into the application source code to implement *AMSE*. In the end, the reflectively injected logging code again uses reflection to access meta-information for controlling the logging mechanism at run-time.



Target Application Source Code Header Files
Reflex
Introspection
Introspective Meta-Information Files
Target Application Source Code Implementation Files
RefLog Binaries
Compiler
Compilation
Annotation Scanning
Reflective Weaving Meta-Information File
Code Weaving
AspectC++
Reflective Logging Deduction
RefLogExec
Introspective Meta-Information Binaries
Compiling
Linking
Compiler & Linker
Executable Target Application Binary
Reflective Logging Enriched Target Application Source Code

Figure 4.1: Reflective Logging Build Intervention Process

Introspection

The first step of the process is to obtain reflective meta-information from the source code base. All C++ source files (*.h, *.hpp, etc.) containing class or struct definitions are mandatory input objects for this activity. The evaluated C++ reflection tool *Reflex* is fed with the source files and generates intermediate source files (e.g. *_rflx.cc). They contain reflective meta-information for essential introspective procedures. These resulting intermediate source files are in Figure 4.1 named *Introspective Meta-Information Files*. Using a C++ compiler (e.g. *g++*) corresponding intermediate *Introspective Meta-Information Binaries* (e.g. *_rflx.o) are created. They can be linked to various executables to support introspection for the current build process or later at target application run-time.

RefLogExec

A set of fixed binaries from *RefLog* and the *Introspective Meta-Information Binaries* are used as input objects to link an intermediate executable named *RefLogExec*. The mandatory *RefLog* binaries are explained later in this chapter. *RefLogExec* is executed afterwards as part of the ongoing target application build process and can be seen as a dynamically created build-time component of *RefLog*. Within *RefLogExec* most preparatory reflective build-time work for reflective run-time processing is done. It gets obsolete immediately after the build because any modification on the source code base (e.g. annotations, methods, etc.) changes the boundary conditions it operates on. This approach is involved but one of the most effective and efficient ways to analyse meta-information extracted from source code using *Reflex*. This additional level of indirection during target application build-time enables the power to exploit reflection not only at run-time. Also annotation-based configuration of the target application at build-time can be realised.

Annotation Scanning

The evaluation of C++ reflection tools reveals, as part of the feature set of *Reflex*, the concept of annotations. Detailed explanations can be found in Section 3.3. Users of *RefLog* can mark classes or methods within their source code base with annotations. This marking expresses their intention to receive behavioural log information during execution. To support this kind of functionality *RefLogExec* deduces which entities were equipped with meta-information. For these entities corresponding reflective logging code is embedded. At the beginning *Reflex* scans the source code base for known annotations and generates an information structure. This structure can be queried by subsequent procedures for relevant meta-information to implement a variety of functionalities.

Reflective Logging Deduction

Within *RefLogExec* the extracted reflective meta-information from annotated source code is used to deduce which logging mechanisms must be injected into the original source code base. The corresponding logging source code is stored in an intermediate output object (*.ah file) in the following referred to as aspect header file. It contains a set of logging-related aspect and advice definitions, that can be directly handed over to *AspectC++*. *RefLogExec* contains a code generator component *AspectGenerator* explained in more detail later in this chapter. Important to understand at this point is just the fact that reflective meta-information is reflectively transformed to AOP-based instructions.
Logging is often seen as a good use case for applying AOP. Writing aspects and advices suitable for direct use with AOP tools is generally a demanding and annoying task. Programmers rarely exploit this power [69], especially for smaller dimensioned projects and rudimentary program analysis. All things considered, within this activity of ”reflective logging deduction” the unpleasant hand-crafted creation of aspects and advices is avoided. The creation process is fully automatised, abstracted and isolated from the programmer and source code base using another form of configurational build-time reflection. Of course AOP provides much more power than representable with a set of simple annotations, but in many cases ease of use is more significant for widespread adoption of mechanisms.

Code Weaving

At this point of the process build-time, reflective concepts were applied to the source code base. Enough meta-information could be extracted to actually generate and weave the logging code into the original source code base. The corresponding header and implementation files are fed in conjunction with the build-time generated aspect header file to *AspectC++*. The resulting binaries already contain the desired reflective logging code. At this point it gets obvious that some limitations and restrictions are introduced by this process architecture. The application source code base can be equipped with reflective logging in so far as *AspectC++* supports it.

Application Linking

The last activity of the process concerns linking of all necessary binaries to the final executable target application binary. The necessary input objects are the previously compiled source code implementation binaries enriched with reflective logging functionalities. In addition the *Introspective Meta-Information Binaries* are mandatory. At this point the different scopes and use cases for introspective reflection get obvious and undercut the power behind this concept:

- at target application build-time for annotation-based configuration
- at target application run-time for the *HROS* feature

4.2 Composition Architecture

The most important fact to understand about the architecture of *RefLog* is that it cannot be seen as most C++ libraries. Usually they consist of headers files containing the available definitions in conjunction with pre-built binaries finally linked to the target application. As outlined more detailed in Section 4.1 *RefLog* consists of different abstract parts and components. They are used in separate phases of the development and building process of the target application. For this reason, it is not really possible to provide a component or class diagram in the traditional style. In contrast, the description will consider the different mandatory aspects. Supplementary a list of entities is provided for a better overview:

RefLog.hpp C++ header for source code interaction

- Preprocessor declarations for logging statements
- Annotation components for annotating class/struct/method

RefLog intermediate executable for build-time reflection activities

AnnotationScanner component for retrieving information about annotations types

AnnotationPool data-structure for clean memorization of type annotations

AspectGenerator component for generating AOP information for code weaving

ObjectSerializer component for serializing objects of known types

Logger component for handling the actual logging process

MethodSignatureHandler component for method internal log message scope enrichment

Timekeeper component for handling any time related tasks

- **Microchronometer** component for measuring time durations
- **MicroSecondEpochTimestamp** data-structure for storing and adapting timestamps

C++ Preprocessing and Annotation-Based Architecture

From the programming perspective a single header file *RefLog.hpp* is essential for programming with *RefLog*. The file contains a set of preprocessor declarations for the different available logging features. The reason why traditional preprocessing is preferred is that an additional level of indirection is established for controlling behaviour at compile-time using macros. This approach supports varying interpretation of logging source code for different build targets or complete removal from the actually compiled application. In some situations, logging is not only expected to be turned off, but as a whole should be non-existent in the built application binaries. Possible use cases are size reduction of resulting binaries or absolute guarantee that no run-time influence happens through logging. In exceptional scenarios, allocated but never used logger objects are not acceptable and desired to be completely removed from applications. Also executed if-statements for checking whether a log message should be written or not can cause unintentional side effects during program execution.
Many progressive development tools[1] found in other programming languages (e.g. Java, C#, VB.NET, etc.) heavily apply annotation-based concepts. The reason may be that annotations provide an improved way of expressing direct relationships between aspects and entities within the source code. In contrast, separated marking approaches like external configuration files complicate the identification of interrelations. *RefLog.hpp* contains a set of annotations with default valued attributes which can be added to class and method definitions. This extension of source code with meta-information supports configuration of execution logging behaviour within the application. For the practical representation in the source code the more widespread term "tracing" instead of the less established "logging" is used. The annotations described in Table 4.1 and related functionalities are provided by the *AMSE* feature of *RefLog*.
Within our *EXTP* example *TraceConstruction*, *TraceDestruction* and *TraceStruction* can be used to log proper construction and destruction of temporary objects to prevent memory leaks. From the business point of view auditing of trades can be realised by careful logging of construction and destruction of related objects in the application. Also the timestamps and contained values are of interest in this case to support confirmability of every single trade within the system. *Trace* in conjunction with *RefLogScoped* can be used for conventional debugging during development and internal process auditing during productive execution. *RefLogTimeMeasured* can be applied for performance optimization tasks in development. For business, auditing of trade timings for long-term analysis of market movement and reaction strategies is a valuable supported option.

[1]The classification in APIs and framework implementations is omitted

Name	Annotatable Entities	Descriptions
Trace (Severity = ”TRACE”)	Class, Method	Entry, exit and execution time duration of corresponding methods are logged at runtime with provided severity; for annotated classes all methods are traced but definition can be overwritten for specific methods whenever necessary
TraceConstruction (Severity = ”TRACE”)	Class	Entry, exit and execution time duration of constructor calls are traced with the provided severity
TraceDestruction (Severity = ”TRACE”)	Class	Entry, exit and execution time duration of destructor calls are traced with the provided severity
TraceStruction (Severity = ”TRACE”)	Class	Entry, exit and execution time duration of constructor and destructor calls are traced with the provided severity
RefLogScoped ()	Class, Method	Extended scope information is collected on a stack for partial stack traces or method internal log message enrichment; for annotated classes all contained methods are treated like being explicitly annotated but definition can be overwritten for specific methods whenever necessary
RefLogTimeMeasured ()	Class, Method	Execution duration time of annotated methods is logged at run-time; for annotated classes all methods are treated like being explicitly annotated but definition can be overwritten for specific methods whenever necessary

Table 4.1: Annotations for *AMSE* provided by *RefLog*

Build-time Architecture

For building *RefLogExec* the following components are linked together:

- *Introspective Meta-Information Binaries* generated from the target application source code
- *AspectGenerator* (*RefLog* internal component)
- *AnnotationScanner* (*RefLog* internal component)
- *AnnotationPool* (*RefLog* internal data-structure type)

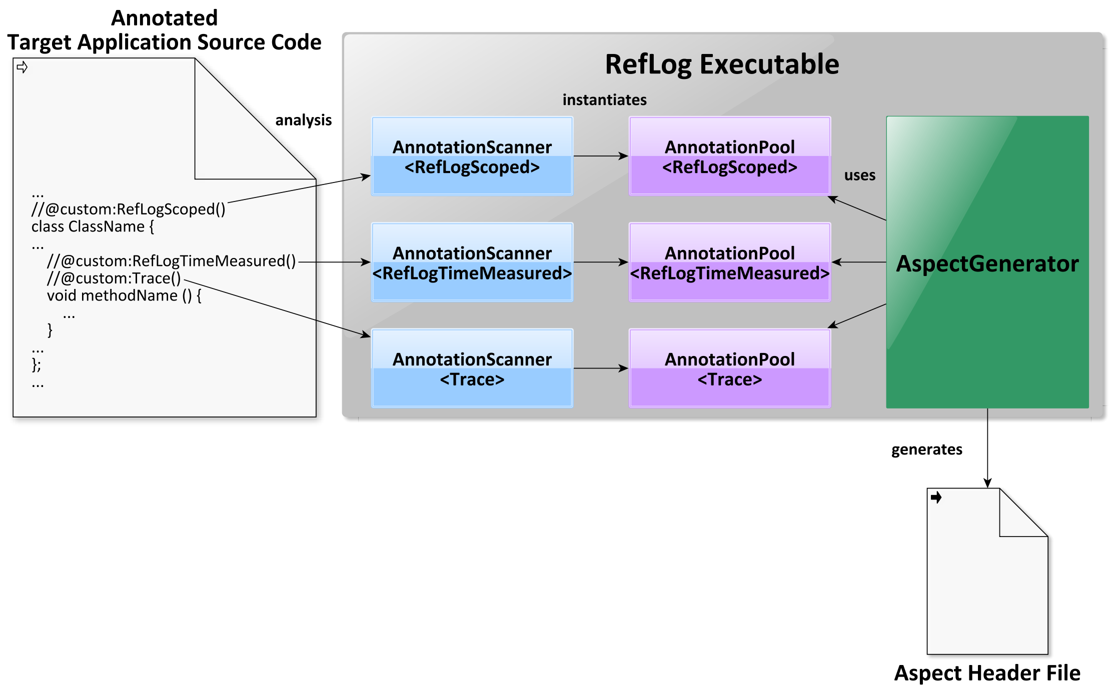

Annotated
Target Application Source Code
...
//@custom:RefLogScoped()
class ClassName {
...
//@custom:RefLogTimeMeasured()
//@custom:Trace()
void methodName () {
...
}
...
};
...
analysis
RefLog Executable
instantiates
AnnotationScanner <RefLogScoped>
AnnotationPool <RefLogScoped>
AnnotationScanner <RefLogTimeMeasured>
AnnotationPool <RefLogTimeMeasured>
AnnotationScanner <Trace>
AnnotationPool <Trace>
uses
AspectGenerator
generates
Aspect Header File

Figure 4.2: Intermediate executable *RefLogExec* analysis and processing

For the following explanation please refer to Figure 4.2. Run-time type introspection provided by *Reflex* allows the *AnnotationScanner* component to scan annotations in the target application source code base. This processing is done during execution of the *RefLogExec* executable at target application built-time. The extracted information is stored in per-annotation-separate instances of *AnnotationPool*. These pools afterwards contain configuration information for the intended logging behaviour of the currently built target application.
The component *AspectGenerator* processes the annotation information from the pools for all known types and generates a single aspect for the reflective logging. Within this aspect for every method a separate advice is created. These advice definitions contain logging statements according to the configuration provided by the programmer for the method. Some possibilities are entry, exit or time duration log statements or code for extended scope information added to log messages within called methods. An example for such statements is given later because of the strong relation to the implementation perspective.

Run-time Architecture

At run-time of the target application equipped with reflective logging several different components and procedures operate. In the first instance the Singleton core component *Logger* is strongly used. All executed log statements finally result in a call to the log method. The *Logger* can redirect log message output to different sorts of output targets (e.g. console, log files, etc.). Several severity levels are distinguished for advanced filter and prioritization capabilities. Measurement and logging of execution duration times for method calls can be requested using the *RefLogTimeMeasured* annotation on methods or even for complete classes. Measurements are taken with microsecond precision.
The second component provided by *RefLog* in use at run-time is the *ObjectSerializer*. It provides the capability to serialize instances of types for which introspective meta-information is available. Programmers request in the source code base serialization of objects using a C++ preprocessor directive from *RefLog.hpp* described later in detail. *ObjectSerializer* heavily uses the possibilities provided by *Reflex* to implement the *HROS* feature. The implementation recursively iterates the integral parts of a type and serializes the values of the currently processed instance. Since types in C++ can be of manifold sorts, a number of different cases must be considered. To reduce complexity and achieve suitable balance between powerfulness and practicability certain restrictions are declared. The following C++ language constructs are not considered during serialization, but for sure are allowed to appear in reflected types:

- unions
- pointers to class/struct data members
- const (volatile) static data members
- reference data members
- function pointer data members

The reason why these constructs are not supported are in the first place that the used *Reflex* tool introduces limitations to handle them. Secondly they have little to no significance in model types which are usually requested to be serialized. For most practical applications these restrictions do not imply too strong limitations. Contrariwise the following much more relevant C++ language features are fully supported:

- inheritance hierarchies
- fundamental (static) data members
- complex (static) data members
- pointer data members (also several levels of indirection e.g. **, ***, ***)
- array data members (also multidimensional)
- string types const char*, char*, std::string

Termination of the serialization mechanism is guaranteed by recursive reference memorization. Objects with cyclic data dependencies (e.g. self referencing pointers) are tracked, enriched with memory address designators and back referenced when a cycle is detected. This solution ensures that no cyclic endless serializations are possible.
The run-time component *MethodSignatureHandler* stores signatures of annotated methods being reflectively logged. These signatures are used for method internal log statements which are requested to be enriched with scope information. This functionality is fundamental for realizing the *AMSE* features. This approach was chosen, since it is one of the simplest and cleanest solutions for solving the problem of behavioural execution scope deduction (i.e. to obtain location information from the program during execution). In addition also the possibility of deducing partial stack traces during execution arises. The functionality is clearly restricted to the call hierarchy annotated with *RefLogScoped* and especially useful when exceptions are thrown. In most cases not the complete execution stack trace is of interest, but only specific important parts of it. The source of the problem with behavioural execution scope deduction in C++ is that no real language integrated mechanism has been declared. The only assurance given by the C++ standard to identify the currently executed function (i.e. execution scope) during run-time is the function-local predefined variable __func__. Unfortunately, the C++ standard declares the actual form of the variable content as implementation-defined [6] and therefore prevents portable use of this feature across different compilers. Neither reasonable content nor uniqueness of the variable values are forced to be provided by compilers. A generalized and portable mechanism cannot rely on this C++ language immanent feature. Other approaches to exploit run-time information for example with function pointers and address deduction imply much higher complexity and create doubt concerning portability among different platforms.

Auxiliary Components

Absolute time is an essential quantity in the context of software logging since many reactive software systems heavily rely on timing, events and other time critical mechanisms. Temporal ancillary information is in many cases as relevant as the core information itself. For this reason the component *Timekeeper* was introduced providing absolute timestamps at execution time. This functionality is not only relevant for logging to enrich log messages with timestamps, but also for program intrinsic profiling and other necessary relative time duration measurements.
In addition to timestamp inference also duration measurement of procedures is commonly used for logging. The *Microchronometer* component is included in *RefLog* and uses *Timekeeper* to track start and endpoints in time during conventional program execution. *Microchronometer* is one of the key figures during the performance evaluation phase of this work.
The last entity explained within this architectural description is *MicroSecondEpochTimestamp*. It represents a specialized data-structure that holds a single timestamp and supports efficient numerical processing of several instances for calculations through overloaded operators. Also serialization to human-readable string formats is provided.

4.3 Execution Space Architecture

The term execution space means in the context of the present work an abstract circumscribable and designated scope in which machine code is executed and certain form of data access structuring takes place. This definition and the following explanations are based on our experiences. In the last decades parallelisation of computational tasks earned rising attention and lead to the introduction of hierarchical execution spaces with manifold characteristics. Figure 4.3 provides an abstract overview.

Today a growing number of applications for example in the area of HPC (high performance computing) do not run single-threaded on a single processing core. They distribute geographically among different networks, machines, platforms and multi-threaded processing cores. *EXTP* is of course a representative example. Debugging and logging tasks often found in such distributed and parallelized applications create the desire for logging tools taking the different execution spaces into account. Specialized architectural designs are mandatory which support suitable and reasonable logging mechanisms to reveal issues. Identifying and resolving strongly distributed and interrelated errors in such software systems is a demanding task. Logging approaches tailored to these particular needs can be invaluable.

Hierarchy and Identification

Today a thread running within a process on an operating system can be seen as the atomic execution space. Machine code execution can be assumed to take place sequentially such that absolute ordering of events is intrinsically guaranteed. On most platforms threads can usually be identified uniquely with an id, at least within a superordinated process. The next hierarchical execution space level are thread subsuming processes. A process can usually also be uniquely identified using an id within the operating system managing it. The next step in the hierarchy are encapsulated and often even virtualized operating systems subsuming several processes in parallel. Unique identification of executed operating system instances is in many cases only provided in a platform dependent manner and must therefore be specifically tackled. Executed operating system instances can be subsumed by cluster or cloud structures which form the next level in the execution space hierarchy. Unique identification in such environments is usually possible with proprietary mechanisms.

Thread Id Enrichment

The *AMSE* feature addresses exactly the problem of revealing additional information about the scope of log messages. For time limitation reasons within this work just the first level of the execution space hierarchy can be dealt with. So, the origin of messages can at least be matched to actual threads within programs.

Parallel Execution and Synchronization

The support of multi-threading is mandatory for state-of-the-art development tools. This requirement introduces unpleasant consequences for the build-time and run-time architecture of

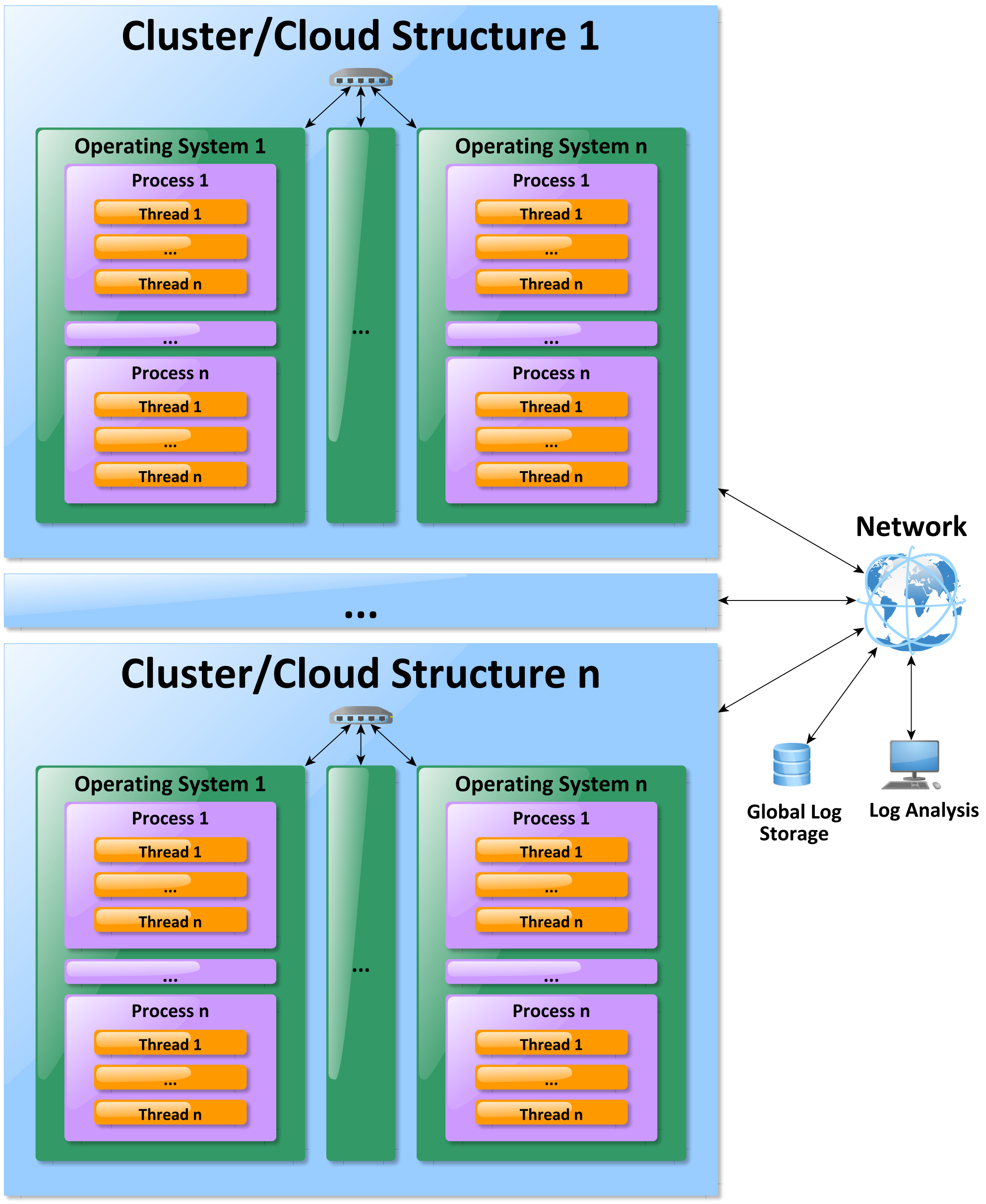


Figure 4.3: Abstracted execution space hierarchy of highly distributed software applications

RefLog. At build-time weaved code must be adapted to suit multi-threading needs and at run-time suitable synchronization of shared resources must be implemented.
When using the *MethodSignatureHandler* component at build-time not only a single stack must be taken into account, but one per thread. Also the registration and identification mechanisms for threads must be realised in a way that the resulting code can deal with it. At run-time access of *MethodSignatureHandler* through different threads must be synchronized. Influencing the ex-

ecution parallelism intended with the use of multi-threading must be minimized. Plain synchronization of the method signature stack handling would result in a non acceptable approximately sequential execution of the target application. The architecture of *MethodSignatureHandler* was chosen to be a map data-structure indexed by thread ids. Map elements reference stack data-structures representing the call hierarchies of method signatures on a per thread basis. The access to this map is synchronized using specific sorts of locks. Since different threads only access the map with their related thread ids, no multiple access to the same value of the map can occur. The only situations introducing synchronization problems are when new threads are added or existing ones destructed within the process. At this time the element structure of the map must be modified. This modification possibly implies race conditions with other threads accessing their related stack instances concurrently. For this reason, two different types of locks are necessary to reach thread-safety for these scenarios:

mutual exclusive lock Only one entity can obtain a lock at the same time and all subsequent lock requests are blocked.

shared lock Several entities can obtain a lock at the same time but only if no *mutual exclusive lock* occurred. For obtaining a *mutual exclusive lock* all shared locks must be unlocked before.

Every value access to the map by thread ids must use *shared locks* and every element structure modification of the map must use *mutual exclusive locks*. For example, mutual exclusive access to the map for adding or removing threads is only granted if no shared lock is active at that time. On the other hand several entities can hold *shared locks* at the same time. This strategy can be seen as multiple read, single write principle [74]. The extension of this general principle is that threads are just prevented from altering the superordinate data-structure (i.e. the map). At the same time they are allowed to access and arbitrarily modify their dedicated stack data-structure indexed by their thread id. Using this approach the required parallel, but thread-safe access to the *MethodSignatureHandler* component can be realized.
At run-time multi-threading introduces another challenge for the *ObjectSerializer* component. Access to the working memory of the process is not guaranteed to be exclusive any more. Multiple processes executed on an operating system instance usually have sealed working memory spaces dedicated for them. These areas are protected by the operating system concerning modifications from outside. No concurrency issues during data access can occur at run-time within a single-threaded program. With multiple threads having the same possibilities to access data within the process working memory this circumstance is obsolete. Serialization of objects using the *ObjectSerializer* can cause concurrent memory access and possibly yields undefined execution behaviour of the program. Unfortunately, the problem cannot be effectively solved within the *RefLog* tool using suitable synchronization mechanisms. The reason is that C++ supports arbitrary memory access using pointer arithmetic at any time. Users can inevitable write source code accessing the same memory areas concurrently. According to our experience and studies no complete and portable mechanism exists to prevent such scenarios. The problem must be solved at the level of application programming. Responsibility for synchronization of possible concurrent access to objects is handed over to the user of the *RefLog* tool. Of course this approach leads to certain inconvenience that must be accepted.

CHAPTER 5

Reflective Logging Implementation

This chapter gives insights into the internals of the *RefLog* implementation and the applied tool set. It is recommended to carefully study and understand the abstract architecture of *RefLog* described in Chapter 4 in advance.

5.1 Tool Set

For this work a large set of tools is used. It supports and simplifies all the necessary and diversified tasks. The different operational areas are described and the corresponding tools enumerated.

Operating System

The whole practical part including programming tasks, application building, *RefLog* benchmarking, reflection tool evaluation, etc. was accomplished within a VMware virtual machine using the freely available VMware Player 6.0.4 build-2249910. The executed virtualized operating system is Ubuntu 14.04 LTS.

Programming

Programming was accomplished on the virtual machine within the cross-platform IDE Eclipse CDT (C/C++ Development Tooling). The source code base for the present work is version controlled using the DVCS (distributed version control system) Git and the related front-end TortoiseGit.

Compilation and Building

- The traditional make is used for controlling and coordinating the necessary build processes.

- Contained programs and scripts from the external reflection tools are used (e.g. genreflex from *Reflex*, ag++ from *AspectC++*, etc.). More information can be found in Chapter 3.
- For compiling and linking C++ source code g++ 4.8.2 from GCC is used.

Benchmarking

For benchmarking *RefLog* GNU time with its verbose option is used, available on most Unix-like systems for measurements on an executed process:

- runtime duration
- maximum working memory consumption

5.2 General Aspects

This chapter explains some important general aspects and details necessary to understand the complete implementation of *RefLog*.

Time

Time is, adjacent to space, one of the two base criteria of computation and needs to be dealt with in this context in a relative and an absolute manner. *RefLog* can be counted to the set of performance critical tools and must treat efficiency and precision as major aspects. A fast and precise mechanism for obtaining time and date information from the execution environment must be chosen. The environment was chosen to be the operating system executing the *RefLog* extended target application in a process.
The relative part of handling time within *RefLog* is concerned with execution duration measurement. Based on our experience the *struct timeval* and the function *gettimeofday* from <*sys/time.h*> are used for the implementation in the class *Timekeeper*. Unfortunately, they are not directly available on all platforms, but usually can be replaced by suitable substitutes. The duration measurement functionality is implemented within the C++ class *Microchronometer*. Timestamp and time duration values are handled in the class *MicroSecondEpochTimestamp*. The entities *Microchronometer* and *MicroSecondEpochTimestamp* are not intended to be thread-safe. Concurrent handling by different threads does not really seem to make a lot of sense. Time duration measurements should be done within a single thread or must be explicitly synchronized in a suitable way. Care must be taken about the added time consumption which is maybe not negligible.
Users usually want log messages to be equipped with human-readable absolute timestamps which inform about the actual point in time something happened during execution. This property is especially important for reactive and geographically distributed software applications as our *EXTP* example. Trading strategies based on arbitrage exploit price differences of identical assets on different markets. When considering geographical distribution of related trading activities, the importance of absolute timestamps for later confirmability of price interrelations becomes clear. For this reason actual time and date information must be efficiently obtained and suitably

converted by logging tools. The functions *localtime* and *strftime* from <*ctime*> are selected for converting timestamps to human-readable formats. Configuration according to different needs can be realised on most platforms.

Annotation Scanning

The target application source code includes *RefLog.hpp*. This header file further includes *Annotations.hpp* containing definitions for all annotations that can be used in conjunction with *RefLog*. For annotations prefixed with "Trace", attributes are chosen to be of type *std::string*.
AnnotationScanner is designed as a C++ template and dedicated template instantiations must be requested for all annotation types. As already mentioned in Chapter 3 the reflection tool *Reflex* fortunately supports extraction and iteration of all reflected types within an analysed source code. This iteration is used to identify all annotated types and methods in a type-safe manner using information returned by *Reflex::Type::Annotations ()*.
The result of annotation scanning is an instance of the C++ template class *AnnotationPool*. It is template instantiated with the same type parameter as the processing template instance of *AnnotationScanner*. This design assures absolute type-safety. *AnnotationPool* instances provide convenient access to all classes and methods which have been annotated with the specific annotation type. Annotation scanning for *RefLog* is designed and structured on a per-annotation-separated basis (i.e. scanner with related storage object) in contrast to a per-type or method basis. In most cases specific features need only a subset or even a single annotation from all possible ones, therefore the chosen approach fits best.

Build-time Invasion Control of Logging Code

To solve the problem of dynamic insertion and removal of specific functional aspects in software applications, it is important to support sufficient power through their foundational architecture. For this reason *RefLog* and the underlying reflective approach was designed in a way to reduce source code intrusion to the inclusion of a single header file (*RefLog.hpp*). Annotations are interpreted as Standard C++ source code comments during conventional compilation and preprocessing. Logging macros in addition provide maximum flexibility for controlling build processes.
RefLog distinguishes five different default severity levels (TRACE, DEBUG, INFO, WARN and ERROR). They can be used to prioritize log messages according to their criticality for debugging and error resolution. Dedicated macros accept any string-based message as parameter and are defined as *REFLOG_LOG_[SEVERITY](message)*.
Also a raw version *REFLOG_LOG(message, severity)* for arbitrary severities is provided. It extends possibilities for using dedicated severities for specific aspects (e.g. synchronization, transactions, etc.) or tracing of special workflows (e.g. complex calculation algorithms, separate business processes). The remaining macros are assumed to be self explanatory and depicted with all others in Table 5.1.

The run-time characteristic of all these macros depends on the build-time invasion and can be actively controlled using specific compiler-passed macro definitions. The different variants of *REFLOG_[SEVERITY]* are used to globally enable reflective logging. In addition the severity-

Name	Descriptions
REFLOG_LOG (message, severity)	Raw imperative logging statement with string based message and arbitrary string based severity parameter
REFLOG_LOG_TRACE (message)	TRACE severity logging statement with string based message
REFLOG_LOG_DEBUG (message)	DEBUG severity logging statement with string based message
REFLOG_LOG_INFO (message)	INFO severity logging statement with string based message
REFLOG_LOG_WARN (message)	WARN severity logging statement with string based message
REFLOG_LOG_ERROR (message)	ERROR severity logging statement with string based message
REFLOG_SERIALIZE (object)	Logging statement for serializing an object of reflectively known type (no C++ fundamental types) to human-readable string format
REFLOG_STACK_TRACE ()	Logging statement for serializing the partial stack trace available to *RefLog* for the corresponding thread to a human-readable string

Table 5.1: Logging macros provided through *RefLog.hpp*

selective compiled-in *REFLOG_LOG* statements are selected (e.g. *REFLOG_WARN* for only ERROR, WARN and proprietary statements, *REFLOG_TRACE* for all statements being embedded in the application). During preprocessing of the target application source code base the macro-based logging statements are resolved based on their severity. So it is possible to inject only log statements with specific severity levels in form of machine code and remove any impact from lower granularity levels at run-time. Resolution of macro-based logging statements during preprocessing can be of the form necessary for expedient logging or elimination.

- With activated logging, the statements are consequently compiled to target application immanent machine code. Reflective logging is possible during target application execution.
- For elimination, the statements are reduced to their neutral representation. The neutral representation of macros is simply nothing (nil). So no compile invasion of logging code takes place except for *REFLOG_SERIALIZE(object)* and *REFLOG_STACK_TRACE()*. These two macros must be transformed to their neutral representation of an empty string *std::string("")*. This is mandatory to prevent source code invalidation through use of this logging statement within other statements or expressions, for example:
  - function1(REFLOG_SERIALIZE(object))
  - REFLOG_SERIALIZE(object).size()

Target Application Build Integration

Insights into the build intervention process of target applications using *RefLog* are provided in Figure 4.1. In this section a stepwise prototypical implementation of this process in form of a *Makefile* is described for *make*.

1. At first the exertion of *Reflex* is presented. The subset of files containing class/struct definitions from the target application source code must be identified. They are separately fed to the python script *genreflex* in the form *genreflex headerFile1.hpp -o headerFile1_rflx.cc*. Within a *Makefile* automatisation can be reached using a suitable *pattern rule*. The generated files (*_rflx.cc) are compiled with an arbitrary C++ compiler (e.g. *g++*), including header files from *Reflex*. The resulting linkable binary files (*_rflx.o) contain introspective meta-information mandatory in later steps. The practical challenge of this part is only the identification of all source files containing type definitions especially in huge and complicated projects. A simple but often inefficient possibility is to introspect all source files within the target application project.

2. The next step is to build the intermediate executable binary *RefLogExec* from the compiled *RefLog* source code with an arbitrary C++ compiler (e.g. *g++*). Afterwards they are linked with the previously prepared binaries (*_rflx.o) and the *Reflex* related static library *libReflex.so*. For successful compilation the *Reflex* header files must also be included.

3. After successful build-time compilation the intermediate executable *RefLogExec* is executed once to generate the aspect header file (Aspect.ah).

4. This file is then used with the application *ag++* from the *AspectC++* tool to generate a logging-code-enriched version of the target application source code. The resulting generated code is further compiled to a linkable target application binary.

5. The last step missing, is to link the introspective meta-information binaries (*_rflx.o) with the previously created target application binary.

For multi-threading applications it is important to understand that the *AspectC++* related program *ag++* cannot be treated as a conventional GCC compiler. It does not provide the command line arguments and options which are available for *g++* or *gcc*. For this reason, whenever weaved source files are compiled using *ag++*, enabling multi-threading support through the command line argument *-pthread* does not succeed. One possibility to enable multi-threading support is to directly link the related posix library using the -l argument like *-lpthread*.

5.3 Automatic Message Scope Enrichment

The architecture description in Chapter 4 explains the relevance of differentiation between build-time and run-time dimension of *RefLog*. The *AMSE* feature relies on configuration through several different annotations for controlling the details at build-time. It therefore implies heavy appliance of *AnnotationPool* instances at build-time in the dedicated C++ class *AspectGenerator*.

This approach is mandatory to deduce where and which extensional logging source code must be weaved into the original application source code base.

Aspect Generation

The *AspectGenerator* component is only instantiated and executed at target application build-time. It is responsible for dynamically generating a single aspect header file (Aspect.ah). This file is only temporarily valid and available for the current target application build. The aspect header file contains a fixed prolog and epilog. The prolog is depicted in Figure 5.1 and declares the following parts:

- guard macro definition (definition closed in epilog)
- header include for using the introspective reflection mechanisms from the *Reflex* tool also for the weaved source code at target application run-time
- header include of *RefLog.hpp* for accessing the global entity *MethodSignatureHandler* explained in detail later in this chapter
- definition begin of the *RefLogAspect* aspect enclosing all the method corresponding advices for weaved source code (definition closed in epilog)

```
#ifndef _REFLOG_GENERATED_ASPECTS_AH_
#define _REFLOG_GENERATED_ASPECTS_AH_

#include <iostream>

#include "Reflex.h"
#include "../RefLog/RefLog.hpp"


aspect RefLogAspect {
```

Figure 5.1: Build-time generated aspect header file prolog

The epilog is of minor interest. It just contains the closing fragments of the *RefLogAspect* aspect and guard macro scopes introduced in the prolog.
The main part enclosed by prolog and epilog contains the definition of advices. Since logging can be configured differently for every single method, the weaved source code can therefore possibly deviate for all of them. The generated advice decomposition was chosen such that for every method a dedicated around-invoke advice is introduced. A pointcut expression identifies just one specific method. This design assures that the generated source code deduced from the information provided through annotations is weaved in only at the correct join points (i.e. every execution of the method within the target application source code base). For a more detailed explanation of the AOP concept and its mechanisms and possibilities provided trough *AspectC++*

```
advice execution ( "void extp::trade::Stock::postOrder(...)" ) : around () {

    // RefLogScoped
    RefLog::methodSignatureChannel.pushMethodSignature ( JoinPoint::signature () );

    // Trace
    REFLOG_LOG( "Calling " + std::string ( JoinPoint::signature () ), "TRACE")

    // RefLogTimeMeasured
    RefLog::Microchronometer microchronometer;

    // join point (postOrder method execution)
    tjp->proceed();

    // RefLogTimeMeasured
    const RefLog::MicroSecondEpochTimestamp& duration = microchronometer.stop ();

    // Trace
    REFLOG_LOG( "Called " + std::string ( JoinPoint::signature () ), "TRACE")

    // RefLogTimeMeasured
    REFLOG_LOG( "Execution duration of " +
        std::string ( JoinPoint::signature () ) +
        " was " + duration.duration (), "TRACE")

    // RefLogScoped
    RefLog::methodSignatureChannel.popMethodSignature ( "" );
}
```

Figure 5.2: Example advice definition for weaved source code

please refer to the tool related documentation [61]. To clarify the outlined description an example advice including weaved source code is given in Figure 5.2.

This advice is defined with the pointcut identified by the pointcut expression *void extp::trade::Stock::postOrder(...)*. Just the method *postOrder* with return type *void* and any parameters in the class *Stock* from namespace *extp::trade* matches. In this case the parameters of the method are not of relevance since there is only one method named *postOrder* within the class. It can be observed that an around-invoke-advice is used. The statement *tjp->proceed();* does not really belong to the weaved source code, but represents the actual process accomplished at the join point (i.e. execution of the method *void extp::trade::Stock::postOrder(...)*). The statements in front of *tjp->proceed();* are executed immediately after the method call, but before the first instruction of the method. The reason for this behaviour is the advice ”execution” declaration. The statements after *tjp->proceed();* are executed after the last instruction of the method but immediately before the return of the method to the call point. In contrast to around-invoke-advices also before- and after-advices can be defined. They are useful for tasks, that need to be weaved either only before the join points or afterwards. The example from *EXTP* presented in Figure 5.2 shows all possible reflective extensions with their related annotations. Not all of the presented source code parts are always weaved in. It depends on the annotation-based logging configuration of the classes and methods in the target application source code base.

Method Entry and Exit Logging

Annotations prefixed with the term *Trace* are intended to serve as markers and configuration mechanisms for automatic advanced logging of method calls. After annotation scanning, the necessary information is available and accessed by *AspectGenerator* to generate log statements for all join points (i.e. method calls). Either complete classes or just specific methods can be annotated. If both entities are annotated simultaneously with possibly different attribute values, the more specific definition of the method is preferred. Generic advices for non specifically annotated methods of a class are generated in a first iteration, the remaining ones for specific methods in a second one.
The resulting source code fragments in the advice are *REFLOG_LOG* logging statements. String-based scope information provided by *AspectC++* with *JoinPoint::signature ()* is used before and after the join point action. The applied severities are obtained from the corresponding "Severity" annotation attribute. An example is given in Figure 5.2.

Execution Time Logging

The annotation *RefLogTimeMeasured* can used by programmers to configure *RefLog* to reflectively deduce the execution duration time of method calls. Measured results are output with corresponding messages to the configured log sink (e.g. log file, console). This message provides human-readable microsecond precise information about the time duration of a specific method call. The functionality is implemented as all scope enriching ones using additional weaved logging source code statements before and after the join point action. In this case an *Microchronometer* instance is constructed and initialized with the current time before method-related code is executed. It is stopped with a call to *MicroSecondEpochTimestamp::stop()* after the method execution has finished. *MicroSecondEpochTimestamp::duration()* immediately returns the desired time duration of the method execution. These reflective code fragments are also contained in the example presented in Figure 5.2.

Method Signature Enrichment of Log Messages

The annotation *RefLogScoped* can be used by programmers to configure *RefLog* to reflectively deduce and prepare advanced scoping information for log messages. Logging statements within the entered scope (e.g. a log message within a method) is then enriched with the desired scope information (e.g. a method signature, class name, etc.). The origin of log messages within the target application source code base can so effectively be deduced.
The core component for dealing with scope tracking and stack trace preservation is the component *MethodSignatureHandler* already mentioned in Chapter 4. It administrates a state consisting of two parts:

- A thread identifier indexed map of stacks for storing method calls on a per thread basis.

- A special form of mutex (*boost::shared_mutex*) [74] for implementing the synchronization strategy already explained in Subsection 4.3.

Calls to *boost::shared_mutex::lock ()* and *boost::shared_mutex::unlock ()* are used for mutually exclusively modifying the map elements to support insertion of newly started threads. For thread intrinsic and encapsulated manipulations of the related stack elements the following shared lock/unlock variants are used.

- *boost::shared_mutex::lock_shared()*
- *boost::shared_mutex::unlock_shared()*

They support parallel access to the map instance except for cases in which mutual exclusive access is requested. *MethodSignatureHandler* provides the following methods for concurrent use by several threads:

- *void pushMethodSignature (const char* methodSignature)*
  inserts the passed signature on top of the threads call stack; if no thread entry exists for the thread, a new entry in the stack preserving map is created; used on method call events
- *const char* getMethodSignature ()*
  provides method-internal access to the signature currently on top of the threads call stack; if the method is annotated with *RefLogScoped*, the signature on top conforms to the current execution scope; used in method internal scope enriched log statements
- *void popMethodSignature (const char* methodSignature)*
  removes the passed signature on top of the stack; if the last signature is removed from the stack, the thread is assumed to terminate and therefore precautionary removed from the map to prevent memory leaks; used in method call return events

The thread separated stack instances are designed accessible from the whole application. This design decision is mandatory because method related weaved logging source code must directly operate on the thread specific stack from within any execution scope. The functionality must be available immediately after the application start. An example of weaved in source code that performs the necessary modifications on the stack before and after method-internal statements is given in Figure 5.2.

Partial Stack Tracing

The administrated scope information from *RefLogScoped*-annotated methods can also be used to obtain some kind of partial stack traces at run-time. For this functionality specific boundary conditions must hold as explained in the following. The logging macro *REFLOG_STACK_TRACE()* can be used to retrieve a human-readable string representation of the currently tracked thread-local method call stack. It must be kept in mind that only calls of *RefLogScoped*-annotated methods are taken into account. Even if all classes and consequently all methods in the source code base have been annotated, possible global function calls are not considered. So no complete stack trace can be provided because at least the root function call to *main* is missing. Although completeness cannot be expected, it is necessary in very seldom cases. The preponderant number of scenarios is just concerned with specific areas and pieces of the whole application functionality. In this cases partial stack traces provide much better overview because of their configurable level of precision and volume.

Scope Jumps and Method Calls Through Pointers

The last details considered for functionalities from the *AMSE* feature are scope jumps:

Exceptions In C++ at any point and within any execution scope (i.e. function or method) an exception of arbitrary type can be thrown. The exception handling mechanisms and positions completely depend on the target application source code and cannot be effectively analysed. Each exception thrown beyond the current execution scope (scope jump) invalidates the complete call stack because the displacement cannot be extracted at run-time. *RefLog* recognizes scope jumps from methods through incompatible signatures in *void popMethodSignature (const char* methodSignature).* As a consequence an entire unwind of the threads call stack is initiated. From this time on, the call stack is preserved in the correct way again, but of course with a completely different and unpredictable starting point. Each execution scope request until the unwind and call stack recovery is completed and results in an empty string return. Call stack recovery happens through consequent function/method invocations building up a new relative call stack. Conventional program execution flow should usually not throw exceptions. It is a well-known pattern to not use exceptions for flow control, but only for real exceptional cases in the application. If programmers for whatever reason use such constructs, the *RefLog*-served partial stack trace is only guaranteed to be valid until the point the exception is thrown. The stack unwinding process to the handling catch-clause cannot be effectively tracked due to technological limitations and restrictions of C++ at run-time. If there is no other possibility to work around exceptions, the stack trace should be extracted within the first possible catch-clause and before any further method call.

Goto-statements Although the use of goto-statements within object-oriented source code is generally not recommended and not the preferred solution they can occur for unforeseen reasons. In general the same problem already explained in conjunction with thrown exceptions also directly applies to program internal jumps as a consequence of goto. Also in this case the *RefLog*-served partial stack trace of the corresponding thread is guaranteed to be valid until the point of goto. It then recovers in the already explained way, but is invalid in the meantime.

Method pointer calls A final limitation concerns method calls of template types, which only work properly with ”call” advices. For successful code weaving all join points must be identified and modified according to the advice definition. The problem in this particular case is that method calls accomplished using class member function pointers cannot be unambiguously resolved. The functionalities from the *AMSE* feature do not work in such situations, but also do not invalidate stack information. The raw logging mechanism without reflective extensions is naturally still guaranteed to be correct.

5.4 Human-Readable Object Serialization

The logging macro *REFLOG_SERIALIZE (object)* represents the target source code intrusive entry point of this feature. It can be used in any expression or statement where instances of *std::string* can be applied. This feature is in contrast to *AMSE* mainly concerned with introspective reflection. *AspectC++* is not necessary for the implementation. This section provides information about implementing object serialization with the functionalities and power from the C++ language and the reflection tool *Reflex*.

Run-time Type Deduction and Serialization Entry Point

At run-time arbitrary instances of reflectively known complex types can be passed to the macro *REFLOG_SERIALIZE(object)*. One problem is how to deduce the type of the passed object and how to pass the object itself in a type-safe manner. Fortunately, the C++ type system contains two very useful mechanisms presented in the following listing which provide in combination with *Reflex* a suitable solution.

- The C++ language operator *typeid* returns a unique identifier of type *std::type_info* for every passed type [6]. These type identifiers are used internally by *Reflex* to administrate all known reflected types in a collection.

- The next important construct is the C++ pointer type *void**. It supports pointing in principal to anywhere in the memory including each instance of an arbitrary type. The immanent problem with this type are dangerous type casts enforced with its use. Usually type safety is surrendered in related program designs. Contrariwise the *void** type provides maximum generality and is therefore an elegant solution to identically treat and pass instances of any type. In general *void** declarations are not recommended and should be omitted as far as possible. In exceptional cases they provide the mandatory powerfulness. In this context it is important to know exactly about the impacts and consequences the use of *void** implies.

Type safety is an aspect of major importance for object-oriented programming and cannot be dispensed by *RefLog*. The two constructs (*void** and type identifiers from typeid) are coupled to form a type-safe type deduction mechanism. According to the preceding explanations the implementation of the logging statement *REFLOG_SERIALIZE(object)* looks as depicted in Figure 5.3. It can be observed that method *static std::string RefLog::ObjectSerializer::toString(void* instance, const std::type_info& typeInfo)* of the C++ template class *ObjectSerializer* is called as the entry point of the serialization process. The passed object address is casted to the type *void** and the corresponding instance of *std::type_info* is directly deduced from the object. The method returns an object of type *std::string* and can therefore be used within expressions and statements. This design prohibits the use of a semicolon to finalize the statement.

```
#define REFLOG_SERIALIZE(object)\
    RefLog::ObjectSerializer::toString ( (void*) &(object), typeid ((object)) )
```

Figure 5.3: Implementation of the logging macro *REFLOG_SERIALIZE(object)*

Serialization Strategy

Serialization is initiated with a static method call to *ObjectSerializer*. An instance of type *void** pointing to the actual serialization subject and corresponding type information in form of an instance of type *std::type_info* are provided. Based on these entities an introspective reflection of the type structure is accomplished with corresponding value deduction from the instance. The feature is implemented with a recursive structural decomposition strategy:

1. At the begin *Reflex* is queried for the introspective meta-information of the currently examined serialized type using the passed type identifier. This reflective type information is only available for reflectively known types within *Reflex*.

2. The returned instance of type *Reflex::Type* supports access to the structural type information necessary for introspection. The returned type is verified to be valid, otherwise the serialization is aborted with the unknown type marker *"?TypeName?[...]"*.

3. In all other cases the recursive structural decompositional introspection and serialization can be initialized. From this time on the *Reflex*-immanent type descriptor *Reflex::Type& type* is used instead of the compiler-deduced one. A stream is recursively passed used throughout the serialization process to collect the resulting information.

4. In this context, recursion is implemented using differently named methods with compatible parameter sets. These serialization methods are later explained in dedicated subsections dealing with the different constructs of the C++ type system. They call each other or even themselves again on different recursion levels during the structural type decomposition. The recursively passed object (*const Reflex::Type& type*) is never modified within the recursion. The pointer to the serialized instance of type *void** dynamically changes and cannot be relied on to always point to the objects start address. For example, in case of array value extraction during recursive method calls the passed address is frequently modified for iterating all array elements.

At the topmost level, structural decomposition of *Reflex::Type& type* takes place using the following methods from *Reflex* [2]:

- *bool Reflex::Type::IsFundamental() const* - Returns true for fundamental C++ types
- *bool Reflex::Type::IsClass/IsStruct() const* - Return true for complex C++ types
- *bool Reflex::Type::IsArray() const* - Returns true for C++ array types
- *bool Reflex::Type::IsPointer() const* - Returns true for some C++ pointer types

Unfortunately it is not directly possible to handle all pointer types using *bool Reflex::Type::IsPointer() const* because of limitations explained later in this chapter. The raw identification of pointer types is accomplished by checking existence of at least one '*' character within the type name.

Fundamental Types

Fundamental types are in this context differentiated in two groups:

1. Fundamental types defined within the C++ standard.

2. Commonly used types for example from the C++ standard library. They are so widespread that special treatment is mandatory. A good example is the type *std::basic_string<T>* (widely used as *std::string*). It is not reflectively serializable with the mechanisms provided by *Reflex* and treated like a special fundamental type within *RefLog*.

Fundamental types should be serialized directly using the provided C++ streaming operators and cannot be directly passed to *REFLOG_SERIALIZE(object)*. They are only handled correctly when used as members of a reflectively known C++ *class* or *struct*.
At the end of the recursive serialization process the passed *void** pointer contains the address of the fundamental type to be serialized. A pointer dereferencing cast of form **reinterpret_cast<T*>* returns the correct value. T represents in this statement the fundamental type (e.g. char, bool, signed long long int, etc.) specified by *const Reflex::Type& type*. Finally the actual fundamental type instance serialization can be done using the standard C++ streaming operator« in a type-safe manner.

Composition and Inheritance

Most objects requested to be serialized are complex types either through composition (data members) or inheritance. Such types are serialized by first being hierarchically and structurally decomposed into the foundational fragments:

1. At first all inherited super types (base classes) of the serialized type are iterated and serialized with the declared inheritance qualifiers (e.g. virtual, public, etc.). Recursive serialization procedures are initiated for them with their corresponding obtained types and adapted memory address pointers to the sub-parts of the instance.

2. After inherited type parts were recursively resolved and serialized the compositional parts are examined. The functionality is realised with an iteration of all data members. The *void** instance pointer again must be adapted for all recursive calls to be correct for the actual data members.

In addition to Listing 5.4 the following different forms of data members must be distinguished for the compositional serialization process [2]:

- *bool Reflex::Type::IsStatic() const* - Returns true for static members
- *bool Reflex::Type::IsEnum() const* - Returns true for enum members

Beside the pure value serialization also the type and member names including possible qualifiers (e.g. virtual, public, const, static, volatile, etc.) are serialized to form a clean and human-readable representation.

Pointer Dereferencing

Pointers are core constructs within the C++ type system and can be used for multifarious utilizations. *RefLog* supports multilevel pointer dereferentiation for data members, but instances of pointer types cannot be passed to *REFLOG_SERIALIZE(object)*. For example, a function local variable *int *pInteger* cannot be serialized with *REFLOG_SERIALIZE(pInteger)*. On the other hand a data member of type *const char****** will be dereferenced to a *const char**. This type is interpreted as a conventional null terminated C-string and can be successfully serialized.
Pointer types (e.g. *T****) can be used for example for reference chains through several function calls or to point to dynamic multi-dimensional arrays. *RefLog* cannot deduce from the type information (i.e. *T****) which intention the declaration follows. For this reason, such pointer types are always interpreted to be simple reference chains to a single element at the end. This element is dereferenced and serialized. Maybe an instance of type *T****** points to a dynamic five-dimensional array. Unfortunately, neither the intention nor the necessary array dimension sizes can be deduced from the type information.
Clean object-oriented design should avoid the use of pointer constructs for such complex data-structures. There exist other more powerful and safe mechanisms (e.g. containers) from the C++ standard library or the C++ boost library [5]. The only exception are absolutely performance critical implementations like *EXTP* for example, in which microseconds can be crucial for success. For these seldom cases human-readable serialization must annoyingly be implemented independently from *RefLog*.
The pointer serialization implementation recursively dereferences instances of pointer types until the element at the end of the chain can be directly accessed. The reflection tool *Reflex* does not really support convenient handling of pointer types. The necessary mechanisms for deducing the level of indirection (number of *) and dereferentiation must be specifically implemented. For the underlying types, to which pointer types point to, serialization is initiated again. The types *char**, *const char**, etc. are identified, interpreted and serialized as null-terminated strings and treated as special kind of fundamental types within *RefLog*.

Multidimensional Arrays

Arrays with compile-time fixed sizes like for example *int array3D[7][5][3]* provide enough contextual type information to support complete serialization. An array dimension is resolved by obtaining its length and iterating all resulting sub-arrays with one less dimension. This procedure is recursively applied until the pure elements are accessible in the last dimension.

Termination and Cyclic Reference Resolution

During dereferentiation of pointer type instances all dereferenced addresses are stored in a data-structure *static std::set<void*> pointerStore* within *ObjectSerializer*. The structure is cleared after each initiated serialization process. This approach is necessary to assure termination of the recursive serialization procedure. In general it is possible that within software applications reasonable data-structures and models arise which contain cyclic references. This scenario must be taken into account and handled correctly. Serialized data-structures are marked with their memory address as program internal unique identifier. Whenever a cyclic reference is recognized after the second iteration just an infinity marker of form *DuplicateOfInstance@0x7fffee924530* is inserted. The marker informs about the abbreviated endless serialization structure. This design proves that the serialization mechanism terminates at least after examination of all possible memory addresses.

CHAPTER 6

Evaluation and Future Work

This chapter provides detailed information about the evaluation of *RefLog* and the obtained results. Existing issues and limitations of the created reflective logging tool are documented. In the end possibilities for further research, improvements and optimizations are pointed out.

6.1 Functional Evaluation

The general approach is explained in Subsection 1.5. A specific program *RefLogTest* contains a set of test cases for evaluating correctness of specific functional details. Real benchmarking is based on two programs to distinguish the important transformative and reactive application domains [65]. Furthermore, applicability of *RefLog* for different kinds of systems is shown. The implemented features *AMSE* and *HROS* are benchmarked separately in the two programs.

TransfBM is the name of the benchmark program representing the transformative application domain. It is dedicated for benchmarking the *AMSE* feature.

ReactBM is the name of the benchmark program representing the reactive application domain. It is dedicated in the first instance for benchmarking the *HROS* feature, but also applies the *AMSE* feature for duration measurement tasks.

Test Cases

The evaluation program *RefLogTest* contains the following test cases:

CompositionTest handling hierarchical composition of data members for *HROS*

CyclicPointerTest handling cyclic pointers for *HROS*

ExceptionStackTraceTest stack tracing with thrown exception for *AMSE*

InheritanceTest serialization of inherited data members for *HROS*

MethodPointerTest logging behaviour with use of method pointers for *AMSE*

MultithreadingTest logging behaviour with parallel threads for *AMSE*

PointerTest handling pointers of various form for *HROS*

Test cases are collectively executed within the program *RefLogTest* producing a log file containing results for all individual cases.

Benchmarking Approach and Environment

Benchmarking of native applications can be accomplished with different approaches. They have specific advantages and drawbacks:

- Direct execution and measurement on machines with operating systems. The advantages are benchmarking under realistic conditions and the simplicity of the approach. The disadvantages are unpredictable and uncontrollable influences from other applications, the operating system or the machine hardware.
- Program transformation and execution on a virtualized environment. The advantage is that benchmark results are completely independent from the hardware and execution environment. The drawback is that transformation of native programs to virtualized environments may not yield representative measurements.

According to the declared aim of practicability for the present work we chose the first approach. To reduce fuzziness, runtime measurements are accomplished several times and summarized using statistics. At the same time a virtual machine is used as benchmarking environment to mitigate influences from different platforms (i.e. hardware and operating systems). During benchmarks the virtual machine is configured to use 4 processing cores from the CPU and to occupy up to 4GB RAM. Further information about the virtual machine environment and applied measurement tools can be found in Section 5.1.
Measurements are accomplished on two different physical machines with different host operating systems to reach certain heterogeneity in the results.

1. Machine (Notebook)

   Microsoft Windows 8.1 Professional 64-Bit-Version (6.3, Build 9600)

   Intel(R) Core(TM) i7-3840QM CPU @ 2.8GHz (4 Cores) with 8GB RAM

   Samsung SSD 840 Pro Series (SATA)

2. Machine (Workstation)

   Microsoft Windows 7 Professional 64-Bit-Version (6.1, Build 7601)

   Intel(R) Core(TM) i5-2500 CPU @ 3.3GHz (4 Cores) with 8GB RAM

   Intel(R) SSD 320 Series (SATA)

Transformative Benchmark for AMSE

The single-threaded program *TransfBM* includes an implementation of the famous quicksort algorithm. It is fed with a file containing 20,000 random numbers in the range of 0 to 1,000,000 obtained from [1]. The sorting algorithm was chosen because of its recursive architecture leading to a complex method call hierarchy. The source code is provided in Appendix A. As can be seen recursively called methods are equipped with the following annotations from the *AMSE* feature:

- RefLogScoped
- Trace
- RefLogTimeMeasured

The benchmark includes all possible reflective extensions to method calls and produces the maximum reflective overhead for reasonable comparison. For this benchmark the complete runtime of the application is measured and all three logging configurations (explained in Subsection 1.5) are applied. The program is executed for each of the three configurations 30 times and three corresponding average runtime durations are calculated (arithmetic mean) as final results. Additionally the size of the binaries and the maximum resident working memory consumption for every configuration is measured.

Reactive Benchmark for HROS

For the multi-threaded program *ReactBM* we refer back to our example *EXTP*. A small part of an asset analysing system often seen in the financial software domain is implemented. Various indicators (e.g. RSI, MACD, etc.) are used to calculate certain quantitative parameters for different types of assets (e.g. stocks, bonds, options...). The obtained parameters are afterwards used to coordinate trading decisions for the tracked assets. As depicted in Figure 6.1 for this benchmark only a single indicator called *AverageIndicator* is implemented. It calculates the average value from all stored historic values of an asset.

Every asset instance is at the beginning equipped with a base value. The reactive behaviour of the system is simulated using *AssetTracker* for starting a dedicated thread for each asset instance. Within this thread a loop is initialized with a predefined number of iterations. For benchmarking 100 iterations are chosen to be suitable. In each iteration the following tasks simulate a conventional asset value movement caused by trading on the related market.

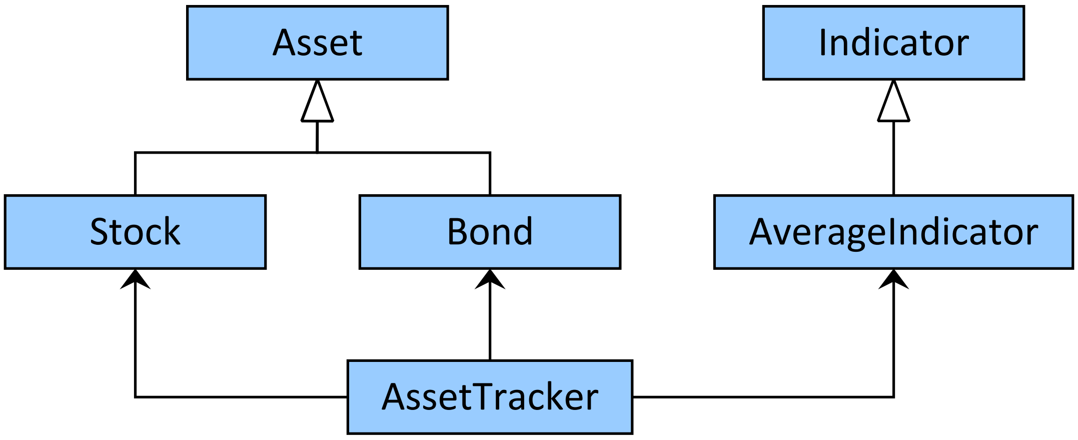


Figure 6.1: Class diagram for reactive benchmark program *ReactBM*

- A random number is generated.
- A deviation to the previous asset value is calculated using the random number to simulate changes on the market.
- Configurable serialization of the asset instance using the *HROS* feature is logged a file.
- The random number is used to generate a varying time delay until the next iteration and change of the asset value occurs.

Because of the reactive nature of the system the overall runtime duration of the program cannot be used for measurement. Several calculation procedure calls interrupted by random waiting periods would not yield reasonable results. Instead execution duration times of specific method calls are measured using the *RefLogTimeMeasured* annotation, program-internally. The execution duration times of calculation method calls to *AverageIndicator* are extracted from the generated log files. These time durations provide reasonable quantitative values for benchmarking the reactive behaviour. The overhead costs are identified again through comparison of separate *ReactBM* builds equipped with two different logging configurations (static logging and reflective logging). Although the principle of these logging configurations is the same, in this case the *RefLog*-internal macro-based configuration cannot be used. Not the complete logging must be configured differently, but only a single serialization statement within the program.

- The first build configuration for benchmarking *ReactBM* includes the serialization statement for the tracked asset instance in every iteration using *REFLOG_SERIALIZE*.
- In the second one the serialization statement is changed to log just a static string of comparable length.

The two finally compared benchmark values are calculated (arithmetic mean) as average execution duration times of the calculation method calls. The comparison provides information about the overhead introduced through adding a single serialization statement for asset object states. It should be noticed that the used asset objects are composed of string-based and numerical data members including an array of fundamental types. So serializing a single state of such an object already produces a substantial amount of logging data. In some cases especially for software auditing in the financial sector such precise object state logging is necessary.

Results

Executed test cases reveal correctness of the *RefLog* functionalities depicted in Table 6.1. The limitations are described in Section 6.3.

Test Case	**Evaluation Result**
CompositionTest	numerical serialization of enums, fundamental type instances, null terminated character strings, std::string instances including qualifiers (const, volatile, mutable), static members, object members of known types are hierarchically resolved and serialized
CyclicPointerTest	cycle pointers are resolved and marked with cycle begin memory address
ExceptionStackTraceTest	stack trace valid until exception is thrown, recovery of destroyed stack information correct
InheritanceTest	inherited data members from base classes hierarchically serialized, also multiple inheritance and all inheritance qualifiers (virtual, public, etc.) supported
MethodPointerTest	stack trace invalidated through method pointer calls but valid again after return
MultithreadingTest	*AMSE* feature also correct with multiple threads
PointerTest	array members also with multiple dimensions, pointer members to single instances also with several levels of dereferentiation, pointer to arrays supported

Table 6.1: Test case evaluation results

Benchmark values are presented in dedicated tables for both benchmark programs. Values for the transformative case are provided in Table 6.2, for the reactive case in Table 6.3.

	Runtime[s]	**Peak Memory[KB]**	**Log Size[MB]**	**Binary Size[KB]**
No Logging				99
Machine 1	0.01	2112	0	
Machine 2	0.01	2112	0	
Static Logging				103
Machine 1	2.65	2116	32.9	
Machine 2	2.54	2116	32.9	
Reflective Logging				351
Machine 1	3.71	2284	32.6	
Machine 2	3.84	2284	32.6	

Table 6.2: Transformative benchmark values from *TransfBM*

	Runtime[us]	**Peak Memory[KB]**	**Log Size[KB]**	**Binary Size[KB]**
Static Logging				379
Machine 1	25.41	2164	797	
Machine 2	44.52	2164	793	
Reflective Logging				375
Machine 1	193.12	2256	735	
Machine 2	417.78	2256	736	

Table 6.3: Reactive benchmark values from *ReactBM*

From these benchmark values evaluation results can be calculated given in Table 6.4. For a better impression on the relative differences the benchmark values are also depicted in dedicated histograms 6.2 and 6.3. The information base allows us to conclude certain facts about *RefLog*:

- *TransfBM* only uses the *AMSE* feature. Approximately one third of the runtime duration is consumed by reflective tasks during application execution. This fraction of resource consumption is according to our defined limit of 50 per cent acceptable. This result gives evidence that reflective extensions can be implemented with sufficiently small runtime overhead for practical applications. The biggest runtime performance deceleration is introduced by writing log messages to a persistent storage (disk). So whenever log files are used as logging sink the additional overhead of reflection introduced by the *AMSE* feature can be accepted.

- *ReactBM* shows that significant overhead is introduced by using the *HROS* feature for serializing objects. The fraction of costs lies in the range of 90 per cent and exceeds our defined limit of 50 per cent. For this reason *HROS* can only be applied in non performance critical situations. It must be noticed that also hand-crafted serialization mechanisms introduce computational overhead. This circumstance relativises the result to some extent.

- The impact of reflection on the working memory consumption is in the range of 4-7% of total costs and therefore far below our 50% limit. For this reason working memory consumption can be seen as less problematic in conjunction with reflection in C++.
- Size of application binaries rises when using reflection because of the additionally linked meta information. This fact can be deduced from the benchmark values of *TransfBM*. The size difference for *ReactBM* arises from inclusion of additional strings for static logging behaviour. Both compiled configurations include reflective mechanisms.

In the context of our research question and aim described in Section 1.4 the following conclusion can be drawn. The question can be answered with ”yes”. The general feasibility of reflective logging (i.e. *AMSE* and *HROS* features) has been proved with the implementation and evaluation of *RefLog*. Resource consumption of working memory could be successfully hold below the set limit of 50%. Concerning runtime this target was only partially reached for the *AMSE* feature, but not for *HROS*.

	Machine 1	**Machine 2**
Runtime Reflection Costs[%] for *TransfBM*	29	34
Runtime Reflection Costs[%] for *ReactBM*	87	89
Runtime Memory Costs[%] for *TransfBM*	7	7
Runtime Memory Costs[%] for *ReactBM*	4	4

Table 6.4: Calculated costs from evaluation results

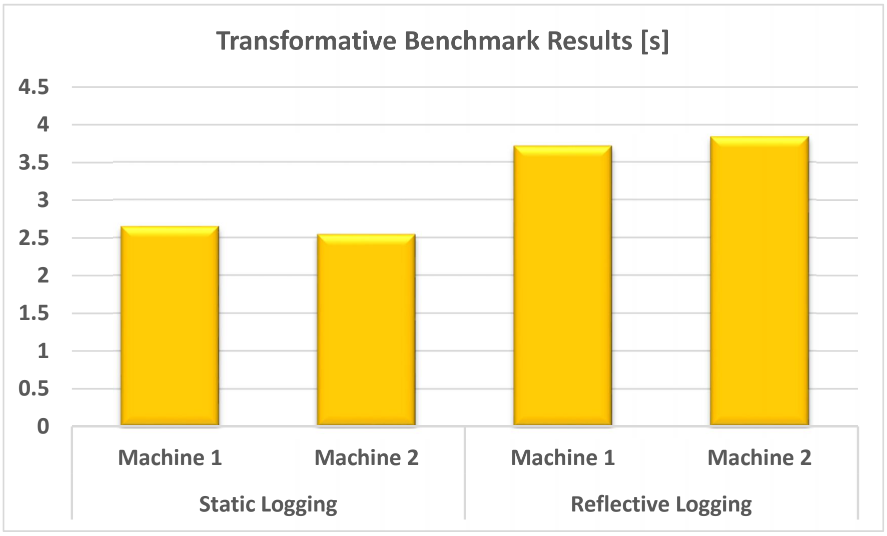


Figure 6.2: Transformative benchmark values for *TransfBM* in relation

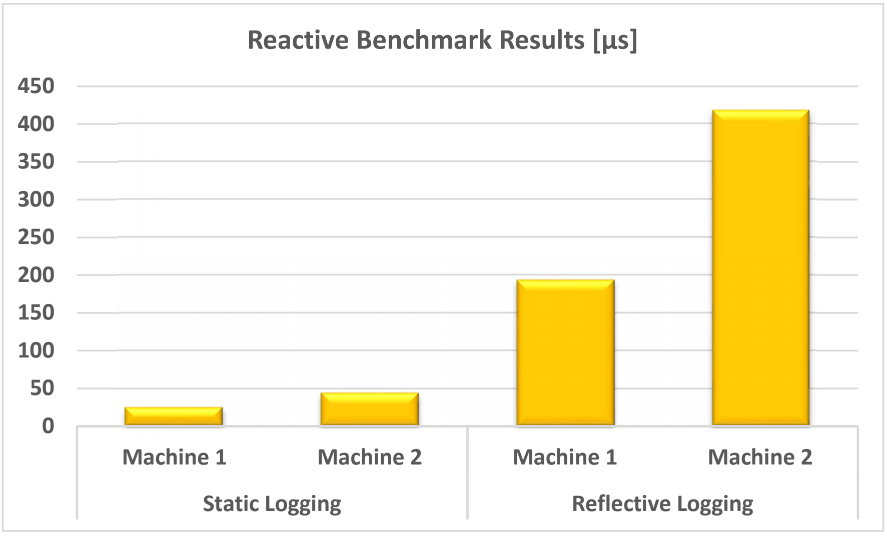


Figure 6.3: Reactive benchmark values for *ReactBM* in relation

6.2 Usability and Integration Evaluation

Next to the functional aspects described in the previous chapter, evaluation of usability and tool integration are important. In the course of evaluating *RefLog* we determined the following aspects. In our opinion they have significant impact on the way the logging tool is used.

Annotation-based Logging

Using annotations to configure logging for an application turned out to be an interesting approach. We think it is the most suitable one for programs with class-based structuring following approved OOP design guidelines. *RefLog* internal annotation types are intended to be used with classes and included methods. The annotation-based approach maybe does not suit certain practically relevant situations:

- Large classes with voluminous methods limit usefulness because scope information gets less valuable. Assistance through extended message scope of logs may be not so helpful to understand a problem.
- Huge projects with collections of many classes being treated in a similar way. Annotations are comfortable for fast adaptions and newly introduced classes. The situation is different whenever large numbers of classes must be reconfigured. They can possibly be spread among several source files too. In such cases all class definitions must be visited and their annotations changed. Such effort is not acceptable and should be avoided. A possible solution is suggested in 6.4.

Object Serialization Format

HROS successfully enables insights into object states at specific execution points during runtime. This kind of information is usually hard to reveal especially in multi-threaded environments and can assist in various situations. The most important C++ constructs for data structuring (composition and inheritance) commonly used in practice are supported. Some drawbacks with the current object serialization implementation were identified during the evaluation phase:

- The format of serialized objects automatically includes memory addresses for back referencing entities in case of cyclic data structures. For most cases the memory information is redundant and can be omitted to improve readability.
- The grade of serialization detail can not be adapted. In some situations only specific object parts are of interest. Serialization range should be configurable using annotations.

Project Integration

The necessary intervention of target application build processes to integrate *RefLog* complicates setup and maintenance of projects. In some cases this intervention may not be possible because of other dependencies or necessary tasks during the build. Runtime duration of builds also increases. This aspect may be unacceptable in some circumstances.

6.3 Open Issues and Limitations

The reflection tools *Reflex* and *AspectC++* provide powerful mechanisms for the most important constructs of the C++ language. Nevertheless, it is of tremendous effort to cope with the language in its entirety. As a consequence many limitations and issues arise as described in this section.

Template Meta-Programming

C++ templates are hard to be handled in a sound and complete manner because of their complexity. A valuable base for appraising the possibilities and impacts of templates to the programming habits in C++ is given by Alexandrescu [9]. He gives a foretaste on the facilities of template meta-programming and shows that debugging such C++ programs, executed at compile-time, is a demanding task. Only information from compiler internal error messages and notifications are provided and they are often difficult to interpret. For this reason *RefLog* does not support advanced logging features assisting in template meta-programming.

Restrictions and Limitations

C++ as a language with a comprehensive historic evolution contains a lot of constructs which lost much of their former relevance (e.g. function pointers, unions, etc.). Some of them are today even frowned upon to be used (e.g. goto statements). Nevertheless, they are part of the C++ standard and must be handled correctly. It can be observed that the reflection tools, used for implementing *RefLog*, even omit direct support of them. For this reason they are also not reflectively logged. Other constructs are also excluded from the implementation for different reasons explained later in this chapter. The following limitations exist for the *HROS* feature:

pointer to members as data members of introspectively reflected types are not serialized

const static data members since their values are well known directly from sources

member references as data members of introspectively reflected types are not serialized

unions cannot be reflected either as type on their own or as data members of reflected types

enum string representation only the numerical values of enum instances can be serialized but not their corresponding string representations

limitations of GCC-XML in general introspective reflection is based on GCC-XML and cannot support additional language constructs

features from c++11, c++14 in general newly introduced features from the latest published C++ standards are not covered by the applied reflection tools and cannot be relied on

Also for the *AMSE* feature restrictions and limitations exist, which have already been discussed to some extend in the previous chapters:

goto statements can invalidate stack hierarchy and therefore break correctness of the feature

exceptions can invalidate stack hierarchy and therefore break correctness of the feature

template class member function pointer calls reflective extensions do not work for calls to template classes using class member function pointers

limitations of ag++ behavioural reflection is based on ag++ and restricted to its powerfulness

features from c++11, c++14 in general newly introduced features from the latest published C++ standards are not covered by the applied reflection tools and cannot be relied on

6.4 Future Work

The scope of this thesis is naturally restricted to the targets described in Chapter 1, but reveals further research topics and directions that can be of interest for future endeavours. Some of the most promising are outlined in the following subsections.

Multiple Platforms

RefLog was foundationally intended and designed for use on different platforms (operating systems, compilers and IDEs). The limited time resources of this work just allowed us to implement and evaluate a single platform solution. In scientific environments in many cases Unix-like operating systems are used, so Linux with g++ from GCC is chosen. It is mandatory to prove support on other platforms with practically representative benchmarks. Only implementations with further compilers and execution on different operating systems can give evidence about real portability characteristics. The most obvious combination would be a verification based on Microsoft Visual C++ on the operating system Microsoft Windows. It is also of high relevance, especially for industrial software development.

Extended Benchmarking

The limited time resources of the present work just allowed us to implement a restricted number of benchmark programs. To provide more insights into resource consumption and performance of *RefLog*, it is necessary to apply the logging tool to many more different programs. They should be chosen from several domains and application fields. Especially in the area of HPC the applicability is of interest.

Advanced Configuration Capabilities

RefLog provides through its annotation-based configuration approach powerful control over its operational behaviour. The use of annotations is according to our experiences quite uncommon in the C++ development domain. Nevertheless, they provide unimagined possibilities for convenient user interaction and maintainable source code integration. Of course, annotations are not suitable for all kinds of configuration requirements and tasks. For this reason, in parallel to the source code intrusive adaption and control mechanisms using annotations, a second servicing point should be established. For example, configuration files or network interfaces. Such extensions raise powerfulness and as a direct consequence usefulness of the development tool for a broader variety of applications. Distributed systems represent a potential domain.

Feature Optimization and Extension

One of the targets of this work is to provide fundamental insights into the topic reflection in the context of C++. Based on these insights the prototypical features implemented in *RefLog* should be optimized and extended in all facets. Also the presented reflective capabilities provided through the available and evaluated reflection tools should be taken into account. Furthermore, completely new features and extensions should be implemented in addition to those presented in this thesis. Features can be identified from software development requirements. Possible candidates are:

manual run-time severity adaptation Severity levels should be adaptable at run-time using for example a network interface to the application.

automatic dynamic run-time severity adaptation With this feature the severity level of written logs should be automatically adapted under specific circumstances. For example a thrown exception should result in detailed log output of the last 1000 messages usually suppressed through static severity level configuration.

automatic parameter serialization For specific method calls the passed parameters and returned result values should be automatically serializable.

instance-based logging Extended logging should not only be configurable on a per-type basis, but also support logging of specific instances of a type. For example, only the interaction with a specific data member should be logged, not all method calls to the type.

extended execution scope enrichment Further levels of execution scope enrichment as described in Section 4.3 should be conceptualized and implemented.

configurable serialization coverage Grade of detail for serialization should be configurable.

Reflection Tool Improvement

The architectural design of *RefLog* should be optimized as a whole for better suitability and to support a broader range of different applications. Beside more and advanced features, also the integration into target applications should be improved to simplify appliance and adoption for arbitrary projects. More powerful or simpler configuration possibilities and chosen default values for existing features can increase their usefulness.

CHAPTER 7

Conclusion

This thesis presented some advanced possibilities for logging in C++ that directly address practical issues and drawbacks found in many software projects. Consequences are inefficient development processes and decreased product quality. Reflection was shown to be a really powerful concept for designing suitable and conveniently applicable mechanisms to overcome these limitations. Many publications mentioned logging and tracing of software applications as an appropriate use case for reflection. In our opinion the key points and really desired features have never been outlined in sufficient detail. The present thesis is one of the first explicitly tackling the topic of reflective logging in C++. Reflection was applied for implementing the desired logging features, but furthermore also for configuring and adapting their detailed behaviour. Not only the pure feasibility perspective received focus. The design also considered the critical aspect of performance in this context. It was demonstrated that reflective solutions for C++ can satisfy acceptable performance criteria. The results are comparable to traditional approaches with all the advantages being offered by reflective extensions.
Another achievement of this thesis was to prove the general applicability and powerfulness of currently available state-of-the-art C++ tools for introspective and behavioural reflection. The C++ language was not equipped with profound reflective features and therefore lots of competing tools were built and published. Some of them are definitely worth to be paid attention. They can offer useful functionalities for a wide variety of elegant applications and problem solutions. The present work showed that reflection is not only practically relevant for creating generic tools on virtualized platforms. It can also be applied in natively compiled environments to combine the benefits of reasonable run-time performance and simplifying abstraction. Hopefully, the results will encourage more scientific investigations. Native reflection use should become common best practice for better software development and resulting products in the future, not only, but also within the domain of the C++ language.

Appendix

Appendix A Transformative Benchmark Program

The following source code is a simple implementation of the famous quicksort algorithm. It is used as transformative benchmark program for evaluation of *RefLog* in Subsection 6.1.

```
#ifndef _QUICK_SORT_HPP_
#define _QUICK_SORT_HPP_

#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <sstream>

#include "../RefLog/RefLog.hpp"

namespace reflog {

/**
 * Simple implementation of the quicksort algorithm for unsigned integers read
 * from a text file.
 */
//@custom:RefLogScoped()
//@custom:TraceConstruction(Severity="DEBUG")
//@custom:TraceDestruction(Severity="DEBUG")
class QuickSort {

    /// data structure for numbers to be sorted
    std::vector < unsigned int > numbers;

public:
```

```
/**
 * Read tab separated unsigend integers from a file also with multiple
 * lines. Assure linefeed on the end of the file to get all numbers.
 */
//@custom:Trace(Severity="INFO")
//@custom:RefLogTimeMeasured()
void readNumbersFromFile ( const std::string& filePath ) {

    // initialize file stream with passed filepath
    std::ifstream fileStream;
    fileStream.open ( filePath.c_str () );

    std::string lineOfNumbers;

    // read in line by line numbers from file
    for ( ;; ) {
        getline ( fileStream, lineOfNumbers );

        if ( fileStream == 0 ) {
            return;
        }

        std::stringstream numberReader ( lineOfNumbers );

        int currentNumber;

        // iterate all numbers of the current line and store them in the
        // internal vector
        for ( ;; ) {
            numberReader >> currentNumber;
            if ( numberReader.eof () || numberReader == 0 ) {
                break;
            }
            numbers.push_back ( currentNumber );
        }
    }
}

/**
 * Root call for quick sorting read in unsigned integers
 */
//@custom:Trace(Severity="INFO")
//@custom:RefLogTimeMeasured()
void sort () {
    quickSort ( 0, numbers.size () );
}
```

```
    //@custom:Trace(Severity="TRACE")
    //@custom:RefLogTimeMeasured()
    void quickSort ( unsigned int l, unsigned int r ) {
        unsigned int pivot;

        if ( l < r ) {
            pivot = partition ( l, r );
            quickSort ( l, pivot );
            quickSort ( pivot + 1, r );
        }
    }

    //@custom:Trace(Severity="TRACE")
    //@custom:RefLogTimeMeasured()
    int partition ( unsigned int l, unsigned int r ) {
        unsigned int pivot = l;

        for (unsigned int i = l + 1; i < r; ++i ) {
             if ( numbers[i] <= numbers[l] ) {
                ++pivot;
                std::swap ( numbers[pivot], numbers[i] );
            }
        }

        std::swap ( numbers[pivot], numbers[l] );

        return pivot;
    }

    /**
     * Print sorted numbers to console.
     */
    //@custom:Trace(Severity="INFO")
    //@custom:RefLogTimeMeasured()
     void printNumbers () {
         for ( std::vector < int >::const_iterator numberIterator = numbers.begin();
        numberIterator != numbers.end ();
        ++numberIterator ) {
             std::cout << *numberIterator << " ";
        }
        std::cout << std::endl;
    }
};

} // reflog

#endif // _QUICK_SORT_HPP_
```

Figure .1: Source code listing of *RefLog* annotated quicksort

Appendix B Base Criteria Evaluation Results

The following tables present the results of the base criteria evaluation described in Subsection 3.2. Information is provided separately for the different reflection tools.

Criteria	Evaluation Result
Powerfulness	introspection, signal and slot
Portability	cross-platform functionality even for mobile and embedded platforms
Documentation	deep and complete documentation
Independence	intrusive using manual macro definitions for meta-data creation but source is generated by moc
Maintenance	UF: frequently/LU: 2014-09-15/ QT since 1991 23years
Maturity	Mature
Intention	Not special
License	GPL for open-source parts but also commercial license available

Table .1: QT's meta-object compiler and system

Criteria	Evaluation Result
Powerfulness	introspection (inheritance, member access and member slicing) support for templates and annotations
Portability	Linux, Windows MSVC >= 7.1, MacOS >= 10.4, Solaris
Documentation	complete reference and source code examples
Independence	non-intrusive except for annotations with fully automatic meta-data generation using genreflex and GCC-XML tool
Maintenance	UF: frequent until 2010 but further maintained in context of ROOT framework/ LU: 2013-01-18/ since 2003 11years
Maturity	Mature
Intention	store huge data proportions for HEP projects like LHC at Cern
License	GPL for open-source parts but also commercial license available

Table .2: ROOT Reflex Library

Criteria	**Evaluation Result**
Powerfulness	introspection, serialization, script binding, callbacks, signal and slot
Portability	gcc and MSVC
Documentation	detailed complete documentation
Independence	intrusive, meta-data creation manually by programmer for every class, meta-gen for generation from doxygen xml but not really powerful
Maintenance	UF: monthly/LU: 2014-10-11/ since 2011-11-05 3years
Maturity	Prototype
Intention	general reflection library for C++ but started from gaming framework
License	Apache License, Version 2.0

Table .3: cpgf

Criteria	**Evaluation Result**
Powerfulness	introspective (inheritance and member serialization), no templates supported
Portability	Visual C++ 6.0 (Service Pack 6) limited, Visual C++ 7.1 and GCC 3.2
Documentation	features completely targeted with source extensions and comprehensive example sources
Independence	source intrusive with macro declarations and method definitions(getClassName), Boost and Loki headers necessary
Maintenance	UF: Unknown/LU: 2004-09-13/Age: 10years
Maturity	Prototype
Intention	Tracing Utility
License	free to use without fee but with copyright notice on all copies and support

Table .4: Arne Adams - A Reflection Library

Criteria	**Evaluation Result**
Powerfulness	introspection (inheritance and member serialization)
Portability	According to source included platform.hpp many platforms and environments (Linux/Windows/Solaris...)
Documentation	Paper 6 pages and 1 basic HTML site
Independence	code generation to add meta-data, so C++ Standard compliant, non-intrusive and auto-generated
Maintenance	UF: Reflex mentions that sourceforge claimed that obsolete/LU: source package 2007-10-25/Age: 7years
Maturity	Prototype
Intention	for PSE (Problem Solving Environments)
License	open-source, free of charge but copyright must be passed with all copies, no warranty

Table .5: XCppRefl - C++ Reflection Library

Criteria	**Evaluation Result**
Powerfulness	introspection (members and method invocation)
Portability	MS Windows Visual C++, Linux GCC, MAC OS X GCC
Documentation	Wiki with much information about general and detailed aspects
Independence	intrusive with manual binding
Maintenance	UF: unknown/LU: 2014-09-10 (License), 2010-09-01 (Source)/Age: at least 5years
Maturity	Prototype
Intention	create reflective library for advanced tools (script binding with Python)
License	open-source, license by Tegesoft obsolete, GNU/LGPL v3 license

Table .6: CAMP

Criteria	**Evaluation Result**
Powerfulness	serialization of types including C++ STL container
Portability	MS Windows Visual C++, Linux GCC + GNU make, is claimed to run with any ISO standard C++ compiler
Documentation	Complete documentation in HTML and Doxygen API
Independence	partially intrusive (CLASSDESC_ACCESS...), auto-generation by preprocessor analysis without pointers with *dump* a human-readable serialization engine for C++ objects
Maintenance	UF: every some months/LU: 2014-09-10/ 10years
Maturity	Prototype
Intention	build general C++ library, focus clearly on serialization (pack)
License	open-source, free of charge but copyright must be passed with all copies, no warranty

Table .7: Classdesc

Criteria	Evaluation Result
Powerfulness	introspection (object construction from types only with default constructor, no enums, unions and bit fields and static members supported, only limited handling of virtual functions, no qualifiers, common class must be inherited)
Portability	Explicit different sources for Windows and Unix but auto-generation of descriptors from debug information only with gcc in Unix
Documentation	Doxygen API for library, 1 HTML page with good introductory documentation
Independence	generates type descriptions automatically from debugging information only with gcc in Unix but intrusive with macros for extensions by programmer
Maintenance	UF: unknown/LU: 2006-01-18/at least 8years
Maturity	Experimental
Intention	try possibilities to extract reflection descriptions from debug information
License	open-source, free of charge but copyright must be passed with all copies, no warranty

Table .8: CppReflection

Criteria	Evaluation Result
Powerfulness	introspection (construction, method invocation, slicing members)
Portability	Used in Windows (MSVC), Linux and MAC OSX
Documentation	introductory documentation about appliance and internal structure, thorough advanced information about the topic in general
Independence	intrusive since crcpp_reflect and related mechanisms must be added to the source, but powerful auto-generated descriptions in databases using clang source parsing
Maintenance	UF: some commits per years/LU: 2014-07-30/since 2011-07-11 so 3years
Maturity	Prototype
Intention	versionable serialization of objects in games like Splinter Cell for example
License	open-source, free of charge but copyright must be passed with all copies, no warranty

Table .9: clReflect

Criteria	Evaluation Result
Powerfulness	extendable serialization/deserialization of C++ objects, script binding
Portability	Linux, MacOS, Windows packages available and clang compiler is necessary
Documentation	unknown
Independence	intrusive with dependency to cpgf (http://www.cpgf.org/), uses Clang for parsing C++ source code to automatically build meta-information from C++ headers, intrusive since it needs to register meta classes with macros
Maintenance	UF: unknown/LU: website updated on 2014-03-12/since unknown
Maturity	Experimental
Intention	unknown
License	open-source, free of charge but depends on license of cpgf
Note	No sources or binaries available

Table .10: idevkit

Criteria	Evaluation Result
Powerfulness	introspection (constructors, inheritance, members, method invocation, language binding, network communication, manual registration no automatic generation of meta-data
Portability	compiled and tested on Windows with MS VC 12 and on Linux with gcc 4.8.1
Documentation	doxygen documentation complete for provided features
Independence	intrusive for classes with inheritance, reflection information created automatically but in cpp files, only Standard ISO C++ necessary
Maintenance	UF: unknown/LU: 2014-07-13/since 2013-06-27 1year
Maturity	Prototype
Intention	General insights into technological basics
License	MIT license

Table .11: rttr

Criteria	Evaluation Result
Powerfulness	introspection of attributes and modification by class to member pointer ->*
Portability	Windows and Linux sources available
Documentation	Comprehensive details in theory, more an article with sample source
Independence	strongly intrusive since a dedicated MOP but without dependencies
Maintenance	UF: unknown/LU: 2014-07-13/since 2000 14years
Maturity	Experimental
Intention	Support for comprehensive and complex abstract business related models
License	free of charge

Table .12: meta

Criteria	Evaluation Result
Powerfulness	only loading variable values from preference files (very restricted)
Portability	developed for MSVC on Windows
Documentation	only few blog posts
Independence	in principle intrusive but also auto-generating some parts
Maintenance	UF: unknown/LU: 2009-05-05/since 2009 5years
Maturity	Experimental
Intention	reflection for preference loading from files
License	unknown

Table .13: autoreflect

Criteria	Evaluation Result
Powerfulness	SQL query generation and object serialization
Portability	uses typeof so only for gcc in general
Documentation	only small description and few examples in txt
Independence	uses TMP for manual dictionary creation by programmer, not automatised
Maintenance	UF: yearly until 2008/LU: 2008-03-05/ since 2007-05-04 7years
Maturity	Prototype
Intention	SQL query generation for persistence and general object serialization
License	GPLv2

Table .14: crd

Criteria	Evaluation Result
Powerfulness	general introspection, instance construction and destruction for class, method invocation
Portability	available binaries only for gcc but is claimed to run also on other platforms
Documentation	some examples and one html site with general explanations
Independence	non-intrusive auto-generation of dictionary information using GCC-XML, intrusive for private member access as friends
Maintenance	UF: yearly until 2009/LU: 2009-02-14/ since 2007-04-30 7years
Maturity	Prototype
Intention	general reflection for C++
License	GPLv2

Table .15: xrtti

Criteria	**Evaluation Result**
Powerfulness	introspection but only supports basic C++ structures and not in depth
Portability	because only Standard C++ is used potentially any platform is supported
Documentation	basic documentation about the most important aspects available
Independence	intrusive, no external tool, but manual generation of dictionary information
Maintenance	UF: frequently /LU: 2014-09-29/ since 2004 10years
Maturity	Prototype
Intention	equip Helium Game Engine with reflection
License	BSD-style license

Table .16: Reflect

Criteria	**Evaluation Result**
Powerfulness	AOP aspect definition and code weaving, slicing for additional members to classes, introspection (inheritance and members)
Portability	for Linux and Windows explicitely but generally for any platform
Documentation	detailed, complete and updated
Independence	non-intrusive with automatic meta-data creation and code weaving
Maintenance	UF: daily builds/ LU: every day again/since 2001-11-06 13years
Maturity	Mature
Intention	support general AOP for C++ like with AspectJ
License	GPL or commercial with Puma in backend

Table .17: AspectC++

Criteria	**Evaluation Result**
Powerfulness	AOP aspect definition and code weaving, preserves source structure and comments
Portability	claim to be available on all platforms for Standard C++
Documentation	deep and complete documentation
Independence	source intrusive with external tool for analysis and code weaving
Maintenance	UF: unknown/ LU: 2005-11-17/ since 2005 9years
Maturity	Prototype
Intention	AOP tool for several languages
License	GPL

Table .18: XWeaver

Bibliography

[1] randomserver - free random numbers. Online http://www.randomserver.dyndns.org/client/random.php accessed 03/30/2015.

[2] Root reflex api. Online http://seal.web.cern.ch/seal/documents/dictionary/reflex/doxygen/html/index.html accessed 03/21/2015, 2006.

[3] Gcc-xml support bug with gcc 4.9. Online http://www.cmake.org/Bug/print_bug_page.php?bug_id=14912 accessed 03/21/2015, 2014.

[4] idevkit. Online https://sites.google.com/site/idevkit accessed 03/21/2015, 2014.

[5] Boost c++ libraries. Online http://www.boost.org/ accessed 03/21/2015, 2015.

[6] ISO/IEC 14882. International standard information technology programming languages c++, September 2011.

[7] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[8] Arne Adams. A reflection library. Online http://www.arneadams.com/ accessed 10/09/2014, 2004.

[9] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[10] Giuseppe Attardi and Antonio Cisternino. Reflection support by means of template metaprogramming. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, pages 118–127, London, UK, UK, 2001. Springer-Verlag.

[11] Giuseppe Attardi and Antonio Cisternino. Template metaprogramming an object interface to relational tables. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 266–267, London, UK, UK, 2001. Springer-Verlag.

[12] John Batali. Computational introspection. AIM-701, 1983.

[13] Charles Bloom. autoreflect. Online `http://cbloomrants.blogspot.co.at/2009/05/05-05-09-autoreflect.html` accessed 03/21/2015, 2009.

[14] Walter Brown, Philippe Canal, Mark Fischler, Jim Kowalkowski, and Marc Paterno. A case for reflection. *JTC1/SC22/WG21/N1775*, January 2005.

[15] Remi Chateauneu. crd. Online `http://sourceforge.net/projects/crd/` accessed 03/21/2015, 2008.

[16] Matus Chochlik. Portable reflection for c++ with mirror. JIOS, VOL. 36, NO. 1, pages 13–26, 2012.

[17] Tyng-Ruey Chuang, Chuan-Chieh Jung, Wen-Min Kuan, and Y. S. Kuo. Objectstream: Generating stream-based object i/o for c++. In *Proceedings of the Technology of Object-Oriented Languages and Systems-Tools - 24*, TOOLS '97, pages 70–79, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Tyng-Ruey Chuang, Y. S. Kuo, and Chien-Min Wang. Non-intrusive object introspection in c++: Architecture and application. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 312–321, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Tyng-Ruey Chuang, Y. S. Kuo, and Chien-Min Wang. Non-intrusive object introspection in c++. In *Software-Practice and Experience*, pages 191–207, 2002.

[20] Cisco. System log management. Online `http://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucm/service/4_2_3/ccmsrvs/ccmsrvs/sssyslog.html` accessed 10/09/2014.

[21] Microsoft Corporation. Attributes c#. Online `https://msdn.microsoft.com/de-de/library/z0w1kczw.aspx` accessed 03/21/2015.

[22] Maximilien de Bayser and Renato Cerqueira. A system for runtime type introspection in c++. In *Proceedings of the 16th Brazilian Conference on Programming Languages*, SBLP'12, pages 102–116, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] Jerry Dennany. Tracelisteners and reflection. Online `http://www.codeproject.com/Articles/3737/TraceListeners-and-Reflection` accessed 10/09/2014.

[24] Tharaka Devadithya, Kenneth Chiu, and Wei Lu. C++ reflection for high performance problem solving environments. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, SpringSim '07, pages 435–440, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[25] Tharaka Devadithya, Kenneth Chiu, and Wei Lu. Xcpprefl - c++ reflection library. Online `http://www.extreme.indiana.edu/reflcpp/` accessed 03/21/2015, 2007.

[26] Gabriel Dos Reis and Bjarne Stroustrup. A principled, complete, and efficient representation of c++. *Mathematics in Computer Science*, 5(3):335–356, 2011.

[27] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, January 1990.

[28] Geoff Evans, Zach Brockway, and Fred Zyda et al. Reflect. Online `https://github.com/HeliumProject/Reflect` accessed 03/21/2015, 2014.

[29] J. Ferber. Computational reflection in class based object-oriented languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 317–326, New York, NY, USA, 1989. ACM.

[30] The Eclipse Foundation. Aspectj. Online `https://eclipse.org/aspectj/` accessed 03/21/2015, 2015.

[31] Inc. Free Software Foundation. Gnu compiler collection. Online `https://gcc.gnu.org/` accessed 03/21/2015, 2015.

[32] Chiclana F.-Carter J. Galli, T. and H. Janicke. Towards introducing execution tracing to software product quality frameworks. In *Acta Polytechnica Hungarica, Vol. 11, No. 3*, pages 5–24, 2014.

[33] P&P Software GmbH. Xweaver. Online `http://www.pnp-software.com/XWeaver/` accessed 03/21/2015, 2008.

[34] Inc. Google. Google. Online `http://www.google.com/` accessed 03/21/2015, 2015.

[35] John A. Interrante and Mark A. Linton. Runtime access to type information in c++. In *In USENIX Proceedings C++ Conference*, pages 233–240. USENIX Association, 1990.

[36] Bryan J. Ischo. xrtti - extended runtime type information for c++. Online `http://www.ischo.com/xrtti/` accessed 03/21/2015, 2009.

[37] Mangesh Kasbekar, Chita R. Das, Shalini Yajnik, Reinhard Klemm, and Yennun Huang. Issues in the design of a reflective library for checkpointing c++ objects. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, SRDS '99, pages 224–233, Washington, DC, USA, 1999. IEEE Computer Society.

[38] J.L. Kenyon, F.C. Harris, and S.M. Dascalu. The c++ hybrid imperative meta-programmer: Chimp. In *International Conference on Computational Intelligence for Modelling Control Automation*, pages 356–361, Dec 2008.

[39] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[40] Brad King. gccxml. Online `http://gccxml.github.io/HTML/Index.html` accessed 03/21/2015, 2015.

[41] Konstantin Knizhnik. Cppreflection. Online `http://www.garret.ru/cppreflection/docs/reflect.html` accessed 03/21/2015, 2006.

[42] Apache Commons Lang. Reflectiontostringbuilder. Online `http://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/builder/ReflectionToStringBuilder.html` accessed 02/27/2015, 2014.

[43] The Qt Company Ltd. Qt's meta-object compiler and system. Online `http://qt-project.org/doc/qt-4.8/moc.html` accessed 03/21/2015, 2015.

[44] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in c++: An operating systems perspective. Technical report, 1992.

[45] Duraid Madina and Russell K. Standish. A system for reflection in C++. *CoRR*, cs.PL/0401024, 2004.

[46] Pattie Maes. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM.

[47] D. Mahrenholz, O. Spinczyk, and W. Schroder-Preikschat. Program instrumentation for debugging and monitoring with aspectc++. In *Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 249–256, 2002.

[48] Axel Menzel. rttr - run time type reflection. Online `http://www.axelmenzel.de/projects/coding/rttr` accessed 03/21/2015, 2014.

[49] Axel Naumann and Philippe Canal. C++ and data. *PoS*, ACAT08:073, 2008.

[50] Oracle. Java annotations. Online `https://docs.oracle.com/javase/tutorial/java/annotations/` accessed 03/21/2015.

[51] Oracle. java.lang.reflection. Online `http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html` accessed 03/21/2015, 2014.

[52] Chen Ping, Cai Xiya, and Jin Yimin. An approach to introduce the reflection to c++. In *Fourteenth Annual International Computer Software and Applications Conference*, COMPSAC 90, pages 52–56. IEEE, 1990.

[53] LLVM Project. Llvm. Online `http://llvm.org/` accessed 03/21/2015, 2015.

[54] Wang Qi. cpgf library. Online `http://www.cpgf.org/` accessed 03/21/2015, 2013.

[55] S Roiser and P Mato. The seal c++ reflection system. In *Computing in High Energy Physics and Nuclear Physics, Interlaken, Switzerland*, pages 437–440, September 2005.

[56] Stefan Roiser. Reflection in c++. Technical report, 2003.

[57] Andrey Semashev. Boost c++ libraries log library. Online `http://www.boost.org/doc/libs/1_56_0/libs/log/doc/html/index.html` accessed 10/09/2014, 2014.

[58] Andrey Semashev. Boost c++ libraries log library attributes. Online `http://www.boost.org/doc/libs/1_56_0/libs/log/doc/html/log/detailed/attributes.html` accessed 10/09/2014, 2014.

[59] Cleiton Santoia Silva and Daniel Auresco. C++ type reflection via variadic template expansion. Online `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3951.pdf` accessed 10/09/2014.

[60] Brian Cantwell Smith. *Procedural Reflection in Programming Languages.* PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[61] Olaf Spinczyk and Daniel Lohmann at pure-systems GmbH. Aspectc++ language quick reference sheet. Online `http://www.aspectc.org/doc/ac-quickref.pdf` accessed 03/21/2015, 2012.

[62] Olaf Spinczyk, Andreas Gal, and Wolfgang 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, 2002. Australian Computer Society, Inc.

[63] Russell Standish. Classdesc. Online `http://classdesc.sf.net/` accessed 03/21/2015, 2014.

[64] Kurt Stephens. Xvf: C++ introspection by extensible visitation. 2003.

[65] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich. Scheduling hardware/software systems using symbolic techniques. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign, 1999. (CODES '99)*, pages 173–177, 1999.

[66] Root Team. Root reflex library. Online `http://root.cern.ch/drupal/content/reflex` accessed 03/21/2015, 2014.

[67] ROOT Team. Root data analysis framework. Online `https://root.cern.ch/drupal/` accessed 03/21/2015, 2015.

[68] Tegesoft. Camp. Online `https://github.com/tegesoft/camp` accessed 03/21/2015, 2015.

[69] Dave A. Thomas. Reflective software engineering - from mops to aosd. *Journal of Object Technology*, 1(4):17–26, 2002.

[70] C. Tull and P. Calafiura. Aspect-oriented extensions to hep frameworks. In *Computing in High Energy Physics and Nuclear Physics 2004*, pages 621–624, 2004.

[71] Matthias Urban, Daniel Lohmann, and Olaf Spinczyk. Puma: An aspect-oriented code analysis and manipulation framework for c and c++. In Shmuel Katz, Mira Mezini, Christine Schwanninger, and Wouter Joosen, editors, *Transactions on Aspect-Oriented Software Development VIII*, volume 6580 of *Lecture Notes in Computer Science*, pages 141–162. Springer Berlin Heidelberg, 2011.

[72] Daveed Vandevoorde. Reflective metaprogramming in c++. Online `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1471.pdf` accessed 10/09/2014.

[73] Detlef Vollmann. meta. Online `http://www.vollmann.com/en/pubs/meta/meta/meta.html` accessed 03/21/2015, 2014.

[74] Anthony Williams. Boost c++ libraries - thread shared mutex. Online `http://www.boost.org/doc/libs/1_41_0/doc/html/thread/synchronization.html#thread.synchronization.mutex_types.shared_mutex` accessed 04/06/2015, 2008.

[75] Don Williamson. clreflect. Online `http://www.donw.org/rfl/` accessed 03/21/2015, 2014.

[76] Edward D. Willink and Vyacheslav B. Muchnick. Preprocessing c++: Meta-class aspects. In *Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems*, page 2, 1999.

[77] Edward D. Willink and Vyacheslav B. Muchnick. Weaving a way past the c++ one definition rule. In *In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, 1999.

[78] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[79] Zhen Yao, Qi-long Zheng, and Guo-liang Chen. Aop++: A generic aspect-oriented programming framework in c++. In *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 94–108. Springer Berlin Heidelberg, 2005.

[80] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the*

Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, pages 143–154, New York, NY, USA, 2010. ACM.

[81] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[82] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 3–14, New York, NY, USA, 2011. ACM.

[83] Istvan Zolyomi and Zoltan Porkolab. Towards a general template introspection library. In Gabor Karsai and Eelco Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 266–282. Springer, 2004.