Knowledge-Based Systems 23 (2010) 704–720

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

AspectC++: An integrated approach for static and dynamic adaptation of system software

Reinhard Tartler [*], Daniel Lohmann, Fabian Scheler, Olaf Spinczyk

Friedrich-Alexander University, Erlangen-Nuremberg, Germany

A R T I C L E   I N F O

Article history:
Received 16 November 2009
Received in revised form 4 February 2010
Accepted 3 March 2010
Available online 10 March 2010

Keywords:
AOP
C++
AspectC++
Programming languages
Adaptable systems

A B S T R A C T

Modern computer systems require an enormous amount of flexibility. This is especially the case in low-level system software, from embedded devices to networking services. From literature and practice, various approaches to modularize and integrate adaptations have been investigated. However, most of this work is implemented with dynamic languages that offer extensive run-time support and enable easy integration of such approaches. System software is written in languages like C or C++ in order to minimize utilization of system resources and maximize efficiency. While for these languages highly optimized and reliable compilers are available, the support for static and dynamic adaptation is rather limited. In order to overcome these limitations, we present an adaptation approach that is based on a sophisticated combination of static and dynamic aspect weaving for aspects written in AspectC++. This facilitates the incremental evolution and deployment of system software that has to be "always on". We demonstrate the feasibility of our approach and its applicability to two pieces of system software, namely the Squid web proxy and the eCos operating system, which is used in the domain of resource-constrained deeply embedded systems.

1. Introduction

Infrastructure software, such as network services or operating systems, is often faced with high availability demands. This poses a real challenge when it comes to deploying adaptations (such as a feature extension or a bug fix) to the running system. Application-specific approaches (such as plugins) to load adaptation modules at run time often provide an extension interface in order to modify the behavior of the base application. However, this approach inherently limits what the adaptation module can change; in fact, it fails for modules that require more flexibility than what the extension interfaces provides. A good example for such an adaptation module is a bugfix, which potentially affects any part of the application.

One of the hardest problems with implementing adaptation modules is to specify *where* to apply *what* changes in the base program. Aspect-oriented programming (AOP) languages provide mechanisms to solve these challenges by *obliviousness* and *quantification* [11]. Obliviousness means that the application of adaptations, called aspects, can be completely oblivious to the component code, in the sense that neither components nor their developers have to be aware of the aspects. Quantification stands for the property that the same aspect code can easily affect several adaptation points.

Most existing AOP approaches can be categorized as either *dynamic* or *static*, referring to the point in time when the actual aspect weaving process is performed. If the aspect weaver performs *static weaving*, the aspects are woven in at compile time, link time, or load time. With *dynamic weaving*, the aspects are woven into an already running program, which promises to overcome the limits that are imposed by traditional extension interfaces.

Dynamic aspect weavers, which feature invasive modification of run-time behavior, are clearly more complex and therefore seldom used than their static counterparts for system software written in C/C++. However, they allow changing the traditional deployment strategy for corrective changes to an already deployed software from having to restart the whole system to a less intrusive processes. In order to bring these benefits to legacy applications, we are looking for an infrastructure that provides enough flexibility for more intrusive modules like bugfix hot-patches or feature extensions. In this article, we present an approach to deploy such adaptation modules flexibly at compile time or run time in low-level system software.

1.1. Problem statement

Ideally, an AOP user would be able to select the aspect language and the weaving approach independently, solely based on

* Corresponding author. Address: University of Erlangen-Nuremberg, Computer Science 4, Martensstr. 1, 91058 Erlangen, Germany. Tel.: +49 9131 8528731.
E-mail address: Reinhard.Tartler@informatik.uni-erlangen.de (R. Tartler).

0950-7051/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.knosys.2010.03.002

the problem to solve. However, most existing aspect languages provide weaver support for *either* static *or* dynamic weaving only. What should be independent in theory, is tightly coupled in practice: the decision for a particular aspect language involves the decision for either dynamic or static weaving as well. From a user's viewpoint, we have de facto "static" and "dynamic" aspect *languages*. This is especially true with languages that are directly compiled into binary machine code. In the C/C++ domain there *are* observable differences in the provided AOP features: The available "dynamic" aspect languages for C/C++ (such as Arachne [9], TinyC^2 [37], TOSKANA [12], or KLASY [36]) offer significantly fewer features than their "static" counterparts (such as AspectC [6], AspectC++ [32], Mirjam/WeaveC [26], *Aspicere* and *Aspicere2* [1,2], and *ACC* [14,15]). Especially language features for generic aspects and static crosscutting are hardly supported. This is unsatisfying; the expressive power of an aspect language (to address the "*what*" part of the problem) should not depend on the intended deployment time (the "*when*") and vice versa. From the viewpoint of weaver implementation, it is, however, understandable: Languages that are strongly based on static typing and compile-time genericity offer hardly any support for run-time reflection, not to speak of means for extension, adaptation, or introduction of new types at runtime. In a sense, Ada, C and C++ are "just not designed" to support many AOP features with runtime weaving. Nevertheless, a uniform, feature-rich, and deployment-time independent aspect language would provide numerous benefits; Section 3 lists some motivating application scenarios.

1.2. Our contribution

We present results from our efforts to add dynamic weaving support to a statically typed and compiled aspect language, for which only static weaving support had existed before. Our approach is based on a novel combination of static and dynamic weaving, which makes it possible to use AspectC++ features such as *generic advice* (statically typed) and *introductions* even for dynamically woven aspects. We are not aware of any other implementation for the C/C++ language domain that offers both, static and dynamic weaving of aspects written in the same aspect language.

Our targeted application domain is applications that run in a resource-constrained environment. For this reason, we cannot afford invasive modifications of the base application, nor a heavy weighted runtime system. Instead we extend our static aspect weaver to collect type information about the adapted software while preparing it for dynamic weaving. This extra information is then used within the C++ template instantiation mechanisms to generate advice code that is executed at runtime.

We analyze and discuss the combination of static and dynamic weaving with respect to two dimensions: language and tools. On the language level, we provide an in-depth analysis of challenging AOP features from the focus of a statically typed base language. On the tool level, we show how we implemented them in a dynamic weaver for AspectC++. Insights about the relationship between static and dynamic weaving on the tool level and an evaluation of our implementation in the context of the Squid web proxy [33] and the eCos operating system [25,10] round up our contribution.

The article extends on previous work, as it provides an actual solution for the problems that have been identified and briefly discussed in [31]. The focus on the implementation challenges of dynamic weaving of static cross-cutting sets it furthermore significantly apart from our previous work on application-tailorable dynamic weaver run-time systems in [13].

1.3. Outline of the article

We begin with a brief introduction into AOP and the AspectC++ language in Section 2 and the presentation of some motivating application scenarios in Section 3. This is followed by the analysis of the implications with respect to dynamic weaving support in Section 4. Section 5 provides an overview of related work. The concepts and some details of our implementation for AspectC++ are described in Sections 6 and 7, followed by two case studies in Section 8. The first one shows how dynamic weaving can help to dynamically adapt a system service – here the proxy server Squid – both at compile time and at run time. The second study demonstrates how our approach has been successfully deployed in the context of deeply embedded systems. Section 9 discusses the pros and cons of our approach. Finally, our work is summarized and some conclusions are given in Section 10.

2. AOP concepts at a glance

Today, most AOP languages use the concepts and terminology that was first introduced by AspectJ[18]. In the remaining parts of this section, we will give a brief overview of the most common AOP language elements in general and the AspectC++ notion in particular, as required for understanding the remaining parts of this article. Even though the introduction is based on AspectC++, it basically holds for any statically woven AOP language.

2.1. Terminology

The most relevant AOP concepts are *join point* and *advice*. An *advice* definition describes a transformation to be performed at specific positions either in the static program structure (*static cross-cutting*) or in the runtime control flow (*dynamic cross-cutting*) of a target program. A *join point* denotes such a specific position in the target program. The target program implicitly exhibits – by its structure and nature – a large set of *potential* join points, which are commonly called *join point shadows* [17,24]. Advice is given by *aspects* to sets of join points called *pointcuts*. Pointcuts are defined declaratively in a *join-point description language*. The sentences of the join-point description language are called *pointcut expressions*. An *aspect* encapsulates a cross-cutting concern and is otherwise very similar to a class. Besides advice definitions, it may contain class-like elements such as methods or state variables.

As an example, Fig. 1 illustrates the syntax of aspects written in AspectC++. The aspect increments the member variable `elements` *after* each call of the function `Queue::enqueue()`. In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. Match expressions are already primitive pointcut expressions and yield a set of *name join points*. Name join-points represent elements of the static program structure such as classes or functions. Technically, match expressions are given as quoted strings that are evaluated against the identifiers of a C++ program. The expression `"% Queue::enqueue(...)"`, for instance, returns a name pointcut containing every (member-) function of the class `Queue` that is called `enqueue`. In the case of overloaded functions with different argument types the expression would match all of them. *Code join points* on the other hand, represent events in the dynamic control flow of a program, such as the execution of a function. Code pointcuts are retrieved by feeding name pointcuts into certain pointcut functions such as `call()` or `execution()`. The pointcut expression `call("% Queue::enqueue(...)")`, for instance, yields all events in the dynamic control flow where a function `Queue::enqueue` is about to be called.

As pointcuts are described declaratively, the target code itself has not to be prepared or instrumented to be affected by aspects.