Chapter 4
Dynamic Reconfiguration of Component-Based Systems: A Feature Reification Approach

Tony Clark
Middlesex University, UK

Balbir S. Barn
Middlesex University, UK

Vinay Kulkarni
Tata Research Development and Design Centre, India

ABSTRACT

Component-based approaches generalize basic object-oriented implementations by allowing large collections of objects to be grouped together and externalized in terms of public interfaces. A typical component-based system will include a large number of interacting components. Service-Oriented Architecture (SOA) organizes a system in terms of components that communicate via services. Components publish services that they implement as business processes. Consequently, a change to a single component can have a ripple effect on the service-driven system. Component reconfiguration is motivated by the need to evolve the component architecture and can take a number of forms. The authors define a dynamic architecture as one that supports changing the behavior and topology of existing components without stopping, updating, and redeploying the system. This chapter addresses the problem of dynamic reconfiguration of component-based architectures. It proposes a reification approach that represents key features of a language in data, so that a system can reason and dynamically modify aspects of it. The approach is described in terms of a new language called μLEAP and validated by implementing a simple case study.

DOI: 10.4018/978-1-4666-6178-3.ch004

INTRODUCTION

Modern software systems are often organized in terms of components. *Component-based* approaches generalize basic object-oriented implementations by allowing large collections of objects to be grouped together and externalized in terms of public interfaces. Such systems execute in terms of messages between components where the distance between message source and target is completely arbitrary.

Once defined, components are instantiated and deployed on platforms that handle the initiation, scheduling, addressing, and message routing for component-based execution. The details of component communication in terms of messages must be transparent whether it takes place on the same platform, on the same network, or over significant geographic distances. Different styles of message passing and component organization lead to different types of architecture. In addition, components may be used at all stages in the Enterprise Architecture (EA) design and development process, from business models through models of IT infrastructure to the implementations themselves. The next section provides an overview of approaches and uses of component-based architectures.

Given the issues outlined above, definition and deployment of component-based systems is more complex than straightforward, single processor, object-oriented applications. However, the life-cycles of both types of applications involve change. A traditional, single point-of-entry Java program can be stopped, edited, recompiled, and restarted almost immediately. An update step to a component-based system is more complex. This is partly because of the implementation technologies involved that require more editing and checking. However, a typical component-based system will include a large number of interacting components. Therefore, a change to a single component can have a ripple effect. If a component were to shut down, other components may not be able to operate effectively. It is likely that the owner of the redeployed component will not have control over some of the components that are affected. The issues related to architectural reconfiguration are described in a later section.

Our approach to solving this problem is to reify those aspects of component-based computation that are involved in post-deployment change. The *process of reification* involves representing in data those aspects of an executing system that would otherwise be rendered in program code. Representation in data form ensures that features can be processed, modified, and replaced without changing the program code. Since changing the program code is the key reason for component redeployment, our proposal is that this helps to solve the problem identified above. Though the approach involves a computational overhead, we believe that the overhead should be acceptable and the reification can serve as an initial step towards eventual redeployment. The motivation for the approach is described in a later section by performing domain analysis leading to a proposal for the key features that should be reified and a case study in a following section that can be used to validate the claim.

We have developed a simple component-based language called *μLEAP* to represent the problem and our proposed solution. The language is a simplification of a larger language and associated tool-set called LEAP that has been used to represent and analyze component-based case studies. The features of μLEAP include higher-order components and functions (those that can take one or more components or functions as inputs and output a component or a function) that are used as the basis for the reification approach described above. The implementation of μLEAP that was used for all the examples in this chapter is available[1] to download as Racket source code. The LEAP tool has been used on a variety of case studies. A snapshot of the LEAP tool is available to download[2].

The chapter is organized as follows. The background section covers the basics of service-