Science of Computer Programming 127 (2016) 24–49

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A formal approach for managing component-based architecture evolution

Abderrahman Mokni [a,*], Christelle Urtado [a,*], Sylvain Vauttier [a,*], Marianne Huchard [b], Huaxi Yulin Zhang [c]

[a] LGI2P, Ecole Nationale Supérieure des Mines Alès, Nîmes, France
[b] LIRMM, CNRS and Université de Montpellier, Montpellier, France
[c] Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France

ARTICLE INFO

Article history:
Received 4 February 2015
Received in revised form 4 March 2016
Accepted 8 March 2016
Available online 18 March 2016

Keywords:
Architecture evolution
Architecture analysis
Evolution rules
Formal models
MDE

ABSTRACT

Software architectures are subject to several types of change during the software lifecycle (*e.g.* adding requirements, correcting bugs, enhancing performance). The variety of these changes makes architecture evolution management complex because all architecture descriptions must remain consistent after change. To do so, whatever part of the architectural description they affect, the effects of change have to be propagated to the other parts. The goal of this paper is to provide support for evolving component-based architectures at multiple abstraction levels. Architecture descriptions follow an architectural model named Dedal, the three description levels of which correspond to the three main development steps – specification, implementation and deployment. This paper formalizes an evolution management model that generates evolution plans according to a given architecture change request, thus preserving consistency of architecture descriptions and coherence between them. The approach is implemented as an Eclipse-based tool and validated with three evolution scenarios of a Home Automation Software example.

1. Introduction

Component-based software development (CBSD) promotes a reuse-based approach to defining, implementing and composing loosely coupled independent software components into whole software systems [1]. While component reuse is crucial to shorten large-scale software systems development time, handling evolution in such processes is a significant issue [2]. Indeed, software systems have to evolve to extend their functionalities, correct bugs, improve performance and quality, or adapt to their environment. While unavoidable, software changes may engender several inconsistencies and system dysfunction if not analyzed and handled carefully. In turn, an ill-mastered evolution engenders software degradation, the loss of its evolvability and then its phase-out [3].

A famous problem of software evolution is software architecture erosion [4,5]. It arises when modifications of the software implementation violate the design principles captured by its architecture. To increase confidence in reuse-centered, component-based software systems, all architecture descriptions must remain consistent and coherent with each other after every change.

* Corresponding authors.
E-mail addresses: Abderrahman.Mokni@mines-ales.fr (A. Mokni), Christelle.Urtado@mines-ales.fr (C. Urtado), Sylvain.Vauttier@mines-ales.fr (S. Vauttier), Marianne.Huchard@lirmm.fr (M. Huchard), yulin.zhang@u-picardie.fr (H.Y. Zhang).

http://dx.doi.org/10.1016/j.scico.2016.03.003
0167-6423/

While a lot of work has been dedicated to architectural modeling and evolution, there still is a lack of means and techniques to tackle architectural inconsistencies, and erosion in particular. Indeed, most existing approaches to architecture evolution hardly support the whole life-cycle of component-based software and only enable evolution of early stage models by propagating change impact to runtime models while evolution of runtime models are not fully dealt with, thus increasing the risks of architecture erosion.

This paper proposes an approach and its implementation to automatically manage component-based architecture evolution at multiple abstraction levels in a manner that preserves architecture consistency and coherence all along the software lifecycle. The approach is based on the Dedal [6,7] architectural model that explicitly models architectures at three abstraction levels, each corresponding to one of the three major steps of CBSD – specification, implementation and deployment, thus granting a full evolution management process. Given a change request at any abstraction level, it transforms Dedal models into B formal models to analyze the requested change and generates an evolution plan that guarantees the consistency of architecture descriptions and the coherence between them. The proposed approach is centered on a formal evolution management model that includes the generated B models, the architecture properties to preserve and a set of evolution rules. It is implemented as an Eclipse-based tool that generates B models from diagrammatic Dedal models and uses our specific solver to resolve architecture evolution. The overall approach is illustrated with a Home Automation Software case-study.

The remainder of this paper outlines as follows: Section 2 presents the background of this work. Section 3 presents our proposal to tackle multi-level architecture evolution (*i.e.* the evolution of architecture definitions composed of multiple description levels) while Section 4 presents the implemented tool and experiments on three evolution scenarios. Section 5 discusses related work and finally, Section 6 concludes the paper and discusses future work.

2. Background

Our approach combines the use of Dedal to model software architectures and B to support automated analysis and verification. This section briefly introduces these languages.

2.1. The Dedal architecture model

2.1.1. Component-based software development by reuse

CBSD follows the *reuse-in-the-large* principle. Reusing existing (off-the-shelf) software components [8] therefore becomes the central concern during development. Traditional software development processes cannot be used as is and must be adapted to component reuse [1]. Fig. 1 illustrates our vision of such a development process which is classically divided in two:

- the component development process (referred to as software component development for reuse), which will not be detailed in the sequel. This development process produces components that are stored in repositories for later use by the software development process.
- the software development process (referred to as software development by component reuse) that describes how previously developed software components can be used for software development (and how this reuse impacts the way software is built).

Dedal is a novel architectural model and ADL [6,7] that targets reuse-centered development. It covers the whole software development by component reuse life-cycle. The main idea of Dedal is to build a concrete architecture composed of stored and indexed components that are found in a component repository as candidates to satisfy the design decisions specified in an intended architecture specification. The resulting concrete architecture can then be instantiated and deployed in multiple contexts. Therefore, Dedal proposes a three-step approach for specifying, implementing and deploying software architectures.

2.1.2. Dedal abstraction levels

To illustrate the concepts of Dedal, we propose to model a home automation software (HAS) that manages comfort scenarios, which automatically controls buildings' lighting and heating depending on time and ambient temperature. For this purpose, we propose an architecture with an orchestrator component that interacts with the appropriate devices to implement the desired scenario.

The *abstract architecture specification* is the first level of software architecture descriptions. It is abstract: it represents the architecture as imagined by the architect to meet the requirements of the future software. In Dedal, the architecture specification is composed of component roles, their connections and the expected global behavior. Component roles are abstract and partial component type specifications. Consequently, the provided interfaces of each role are to be connected to compatible required interfaces. Component roles are identified by the architect in order to search for and select corresponding concrete components in the next step. Fig. 2-a shows a possible HAS architecture specification. In this specification, five component roles are identified. A component playing the *HomeOrchestrator* role controls four components playing the *Light*, *Time*, *Thermometer* and *CoolerHeater* roles.