Science of Computer Programming 78 (2013) 1073–1098

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

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

Reconciling run-time evolution and resource-constrained embedded systems through a component-based development framework

Juan F. Navas [a,*,1], Jean-Philippe Babau [b], Jacques Pulou [c]

[a] AREVA R&D - Euriware, Paris, France
[b] Lab-STICC UBO, UEB, Brest, France
[c] Orange Labs, MAPS Research Group, Grenoble, France

ARTICLE INFO

Article history:
Received 12 March 2011
Received in revised form 17 July 2012
Accepted 13 August 2012
Available online 2 October 2012

Keywords:
Embedded software
Components
Optimization
Evolution
Reifications
Architecture

ABSTRACT

This paper deals with the evolution of embedded systems software at run-time. To accomplish such software evolution activities in resource-constrained embedded systems, we propose a component-based, execution time evolution infrastructure, that reconciles richness of evolution alternatives and performance requirements. The proposition is based on fine-grained optimization of embedded components, and on off-site component *reifications* called *mirrors*, which are representations of components that allow us to treat evolution concerns remotely and hence to reduce the memory footprint. An evaluation on a real-world evolution scenario shows the efficiency and relevance of our approach.

1. Introduction

The development of embedded software for resource-constrained and dedicated systems faces complex design challenges. While the stress of delivering low-cost and resource-efficient products is still present, an increasing number of requirements must be integrated. Especially in the non critical software field, time-to-market has become a central issue, as software designers need to satisfy as quickly as possible the users' needs and to integrate new standards and new technologies.

In the context of embedded systems, Component-Based Software Engineering (CBSE) [1] approaches have been adopted at design-time. CBSE brings the benefit of mastering software complexity and variety [2–4], which in turn reduce development cost and accelerate time-to-market. CBSE defines the concept of a software *component*, an independent and arbitrarily grained entity to be deployed in a system, allowing separation of concerns among other benefits [5,6]. In this paper, we follow the client/server model which is suitable to embedded software development, as in [6–9]. A component has a behavior that is described by a piece of code. It exhibits offered and required services which are described by their interface type (e.g. a set of declarations of functions). Composition contracts among components are established using interfaces: a required interface can be bound to an offered interface of the same type.

* Corresponding author.
E-mail addresses: juan.navas@areva.com (J.F. Navas), jean-philippe.babau@univ-brest.fr (J.-P. Babau), jacques.pulou@orange.com (J. Pulou).

1 This work was done while the author was a graduate student in the Lab-STICC UBO, under the support of Orange Labs.

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.08.004

Thus a CBSE-based design helps to produce quickly an efficient and adapted dedicated system which integrates all the required features. A remaining challenge is the ability to change the system behavior after design stage, i.e. software that has been already deployed and/or running software. This is necessary to support several features such as remote administration and maintenance support, bug fixing, parameter tuning, user preferences adaptation, changing operational context management and refinement of behavior, among others. This requirement refers to the general concept of *software evolution*, which is defined as a continued development that changes system functionality or properties [10], and can be referred to as the need of software flexibility at run-time [11].

Recent research results and real-world experiences have proven that architecture-based approaches (CBSE among them) are beneficial when reasoning about software evolution, as Oreizy and Taylor [12,13] noted. These approaches embrace the concepts of architectural styles and software connectors, which provide a framework for manipulating applications at run-time. However, the efforts leading to a high flexibility of the embedded software at run-time, introduce software artifacts whose presence and execution may conflict with resource capacities [14].

This paper targets embedded systems such as dedicated, micro-controller-based systems found in robotics, handled systems and wireless sensor networks. Due to memory size, processor power and battery power limitations, resources usage has to be optimized. And, these systems do not provide enough resources to support a high flexibility at run-time and a complete evolution infrastructure. We believe that, even if the resource capacity constantly increases, a thrifty solution is relevant as required features also increase.

The subject of this paper is the conflict between *richness of evolution* and its induced *performance costs*, in the scope of non critical resource-constrained component-based embedded software. Related works (cf. Section 2) show that the conflict is manifest in all component-based development frameworks. Among the existing approaches, some of them [7,8] focus on producing a globally optimized binary code, consequently limiting system evolution at execution-time as the generated executable code is extremely rigid and because they do not offer fine-grained optimization techniques. The only way to evolve such systems is a full-replacement of binary image, which may be prohibitive in terms of transmission costs and system reliability.

Other component-based approaches impose strong design-time restrictions in terms of execution models, deployment policies and planning of evolution operations. The most frequent one [15–18] is to precisely define the evolution policies of each component at an early stage of its life-cycle, typically at design time, which allow them to optimize the whole system accordingly. However, this solution forbids unplanned evolutions. In cases where embedded software changes can be predicted and hence formally defined in advance, this approach seems appropriate. In other cases, we are focusing on this paper, it may be prohibitive or impossible to predict and integrate all the capabilities of evolution at design-time.

We conclude that it would be useful to reconcile the requirements evolution at run-time for component-based systems, with the constraints regarding physical resources that we find when dealing with embedded software development. This could be attained by deploying the most adequate *evolution infrastructure*, i.e. to deploy the software artifacts that deal with the observation of the system state, the reasoning and the execution of evolution-related activities, in such a way that the richness of evolution vs. induced performance cost trade-off is minimized for a given application. In order to achieve this, we propose first fine-grained optimizations of how component-based architectures are transformed into binary, executable code, and we drive these optimizations from components' evolution needs. We focus on the optimization of the *glue code*, i.e. the software artifacts that reify the components and their composition at execution-time. Optimization of specific components behavior is application-dependent and out of scope of this work. The idea behind this decision is to remain relatively independent of the characteristics of applications, which are very heterogeneous in the field of embedded systems.

Second, we propose a novel approach based on the *reification* notion. A *reification* is a syntax-agnostic representation of a component at a given stage of its life-cycle. The main idea is to consider two execution-time reifications. The first one, called a *mirror* reification, can be seen as an image of the second one, which is the component running inside the embedded device (i.e. the executable binary code). Unlike this latter, a *mirror* reification executes *off-site*, i.e. not in the embedded hardware platform. Consequently, a *mirror* reification may offer richer evolution-related interfaces without affecting memory footprint. In addition, the embedded reification of the component-based architecture may be aggressively optimized, keeping the minimum set of glue code necessary to evolution actions execution. Mirrors also establish links with other life-cycle stage reifications, in order to retrieve information about how embedded reification was designed, compiled and deployed into the execution platform.

By generating embedded and *mirror* components, as well as optimized mechanisms of evolution through optimized evolution infrastructures, we are able to vary the location of the treatment of evolution concerns and achieve higher levels of optimization, reducing performance costs and keeping a rich set of evolution alternatives.

The remainder of this paper is organized as follows: Section 2 presents the study of the related approaches. Section 3 presents the basis of our proposed approach, as well as the architectural and life-cycle models that will be used in our proposition. Main theoretical foundations of *reifications* are presented in Section 4. In Section 5 we define optimization and evolution infrastructures generation policies to produce evolvable applications that comply with physical constraints in embedded software development. Section 6 evaluates the execution-time performance of our propositions, in terms of memory footprint and execution times of evolution actions. Finally, Section 7 concludes our paper and describes our future work.