The Journal of Systems and Software 81 (2008) 2210–2236

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems ☆

Marco Autili [a], Leonardo Mostarda [c], Alfredo Navarra [b], Massimo Tivoli [a,*]

[a] *Dipartimento di Informatica, Università dell'Aquila, Via Coppito, I-67100 L'Aquila, Italy*
[b] *Dipartimento di Matematica e Informatica, Università degli Studi di Perugia, Via Vanvitelli 1, I-06123 Perugia, Italy*
[c] *Department of Computing, Imperial College London, South Kensington Campus, London SW7 2AZ, United Kingdom*

ARTICLE INFO

Article history:
Received 14 February 2007
Received in revised form 1 April 2008
Accepted 4 April 2008
Available online 12 April 2008

Keywords:
Software architecture
Component-based software engineering
Component assembly
Component adaptation

ABSTRACT

Building a distributed system from third-party components introduces a set of problems, mainly related to compatibility and communication. Our existing approach to solve such problems is to build a *centralized adaptor* which restricts the system's behavior to exhibit only *deadlock-free* and *desired interactions*. However, in a distributed environment such an approach is not always suitable. In this paper, we show how to automatically generate a *distributed adaptor* for a set of black-box components. First, by taking into account a specification of the interaction behavior of each component, we synthesize a behavioral model for a centralized *glue adaptor*. Second, from the synthesized adaptor model and a specification of the desired behavior that must be enforced, we generate one *local adaptor* for each component. The local adaptors cooperatively behave as the centralized one restricted with respect to the specified desired interactions.

1. Introduction

Reuse-based software engineering is becoming one of the main development approaches for business and commercial systems. Nowadays, a growing number of software systems are built as a composition of reusable or COTS (*Commercial-Off-The-Shelf*) components and CBSE is a reuse-based approach which addresses the development of such systems.

In an ideal world, component-based systems are assembled by simply connecting together compatible ready-to-use components,[1] that jointly provide the desired functionalities. However, in the practice of software development it turns out that the constituent components often do not perfectly fit together and adaptation is needed to eliminate the resulting mismatches (Becker et al., 2006; Yakimovich et al., 1999; Szyperski, 2004; Horwich, 1990; Yellin and Strom, 1997, 2002; Zaremski and Wing, 1995; Schmidt and Reussner, 2002; Becker et al., 2004). In particular, considering third-party and black-box components makes the problem worse since there is no way to inspect the source code for possibly solving mismatches from inside. In this setting, while assembling a distributed system from a set of black-box components interacting by message passing, the specific problem we want to face concerns how to automatically prevent deadlocking and undesired (externally observable) interactions of the resulting system. A widely used technique to deal with this problem is to use adaptors and interpose them among the components that are being assembled to form the system. The intent of the adaptors is to moderate the external communication of the components in a way that the resulting system is deadlock-free and complies with a desired behavior (*i.e.*, desired sequences of messages exchanged among the components).

Our previous approach (Inverardi and Tivoli, 2003) (implemented in the previous version of our Synthesis tool (Tivoli and Autili, 2006)) is to build a *centralized adaptor* which restricts the system's behavior to exhibit only a set of *deadlock-free* or *desired interactions*. By exploiting an *abstract* and *partial* specification of the global behavior that must be enforced, Synthesis automatically builds such an adaptor. It mediates the interaction among the components by allowing only the desired behavior specified by the assembler (*i.e.*, the Synthesis user) and, simultaneously, avoiding possible deadlocks.

In a distributed environment it is not always possible or convenient to introduce a centralized adaptor. For example, existing distributed systems might not allow the introduction of an additional component (*i.e.*, the adaptor) which coordinates the information flow in a centralized way. Moreover, the coordination of several components might cause loss of information and bottlenecks,

☆ This paper is a revised and extended version of Autili et al. (2006) that has been presented at EWSA2006.
* Corresponding author.
E-mail addresses: marco.autili@di.univaq.it (M. Autili), lmostard@doc.ic.ac.uk (L. Mostarda), navarra@dipmat.unipg.it (A. Navarra), tivoli@di.univaq.it (M. Tivoli).
[1] Hereafter the terms *component* and *component instance* are used interchangeably.

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.04.006

hence slowing down the response time of the centralized adaptor. Conversely, building a distributed adaptor might extend the applicability of the approach to large-scale contexts.

In this paper, we describe our novel approach to the automatic generation of a *distributed adaptor* for a set of black-box components. Given (i) a Labeled Transition System (LTS) (Keller, 1976) specification of the *interaction behavior* (based on message passing[2]) of each component with its "expected environment"[3] and (ii) an LTS-based specification of the *desired behavior* that the system to be composed must exhibit, our approach generates *component local adaptors* (one for each component). These local adaptors suitably communicate in order to avoid possible deadlocks and to enforce the specified desired interaction behavior. They constitute the distributed adaptor for the given set of black-box components.

In Tivoli and Autili (2006) (and references therein), we have shown how it is possible to automatically derive LTS behavioral descriptions by assuming a partial specification of the system to be assembled. In particular, we give a partial specification of the interaction behavior of each component in the form of a *basic Message Sequence Chart* (bMSC) and *high-level MSC* (hMSC) specification (Uchitel et al., 2004; ITU Telecommunication Standardisation Sector, 1996). By applying our implementation of the algorithm described in Uchitel et al. (2004), the partial specification of each component is automatically translated into the corresponding LTS specification. hMSC and bMSC specifications are useful as an input language, since they are commonly used in software development practice. Thus, LTSs can be regarded as an internal specification language.

Starting from the specification of the components' interaction behavior, our approach synthesizes a behavioral model (*i.e.*, an LTS) of a centralized *glue adaptor*. At this stage, the adaptor LTS is built only for modeling, by interleaving, all the possible (externally observable) interactions considering synchronization on common actions, *i.e.*, the send event for a message and the corresponding receive event. It models a dummy routing component and each message it receives is forwarded strictly to the right component.

By taking into account the specification of the desired behavior that the composed system must exhibit, our approach explores the centralized glue adaptor model in order to find those states leading to deadlocks or to undesired behaviors. This process is used to automatically derive the actual code for the set of local adaptors that implement the *correct*[4] and *distributed* version of the centralized adaptor model. It is worth mentioning that the construction of the centralized glue adaptor model is required to deal with deadlocks in a fully-automatic way. Otherwise, in order to avoid the construction of the centralized adaptor, we should make the stronger assumption that the specification of the desired behavior itself ensures deadlock-freeness and it is consistent with respect to the centralized glue adaptor (*i.e.*, the desired behavior can be enforced against the glue adaptor).

The approach presented in this paper has various advantages with respect to the one described in Tivoli and Autili (2006) and Inverardi and Tivoli (2003) concerning the synthesis of centralized adaptors. The most relevant ones are

- no centralized point of information flow exists;
- the degree of parallelism of the system without the adaptor is maintained. Conversely, the approach in Tivoli and Autili (2006) does not permit parallelism since the adaptor is centralized, single-threaded and the communication with it is synchronous;
- all the domain-specific deployment constraints imposed on the adaptor can be removed. In Tivoli and Autili (2006), we applied the synthesis of centralized adaptors to COM/DCOM applications. In this domain, the centralized adaptor and the server components had to be deployed on the same machine. Now, the approach described in this paper allows one to deploy each component (together with its local adaptor) on different machines.

The Synthesis tool has been extended accordingly in order to enable also the distributed implementation of the generated adaptor model. The distributed adaptor is implemented as a set of EJB component wrappers (Autili et al., 2007). Each wrapper is developed by using AspectJ that easily supports the wrapper tasks of intercepting the component messages and correctly coordinating them. Note that AspectJ is only one possible implementation choice.

2. Background notions

This section provides the reader with background concepts, definitions and assumptions needed for a full understanding of our work. Actually, the discussion has been kept as light as possible in order to give a good intuition to the reader without loosing his/her attention. Detailed formalisms and definitions are then referred to Appendixes A and B.

In our context, a distributed system is a network of interacting black-box and ready-to-use components $C = \{C_1, \ldots, C_n\}$ that can be simultaneously executed. Components communicate by message passing. Messages are exchanged by means of communication channels, performing precise communication protocols that specify (in some formalism) the set of all possible message sequences. Note that, dealing with black-box components, communication protocols specify *external* communication among components by the relatively simple nature of the message exchange (and hence by means of send and receive events) rather than *internal* computation within a component. Generally speaking, communication channels can be

- *asynchronous* – no synchronization points exist and message passing never blocks the sender. This implies a potentially unbounded buffer; in practice, a bounded buffer is used and the sender will block when the buffer is full. In this way, a higher degree of parallelism can be achieved because (possibly) the sender never has to wait.
- *synchronous* – message passing uses no buffer and, due to synchronization points, both senders and receivers can block. The term rendezvous is often used to evoke the image of two processes that have to meet at a specific synchronization point.

For the purposes of this work, we model component interaction by assuming that the components to be assembled communicate by means of synchronous communication channels. This is not a limitation since, in practice, by introducing a finite buffer component to decouple message passing, we can simulate a bounded asynchronous system with a synchronous one (Uchitel et al., 2004; Milner, 1989). Obviously, in this case, there is the necessity of explicitly programming the needed buffers by exploiting the native primitives (of the programming language being used) that are provided to support the synchronous communication. It is well known that reasoning (*e.g.*, deadlock prevention) with the presence of unbounded buffers is undecidable (Brand and Zafiropulo, 1983). From a practical point of view, this motivates the reasonable restriction to consider only synchronous systems (or, possibly, bounded asynchronous ones).

[2] Message exchanging can be used for delivering packages of data or for calling remote procedures.

[3] Dealing with third-party and ready-to-use components, the expected environment is actualized at assembly time by the set of all the other components that are being assembled to form the system.

[4] With respect to deadlock-freeness and the specified desired behavior.