Ian Gorton George T. Heineman
Ivica Crnkovic Heinz W. Schmidt
Judith A. Stafford Clemens A. Szyperski
Kurt Wallnau (Eds.)

LNCS 4063

Component-Based Software Engineering

9th International Symposium, CBSE 2006
Västerås, Sweden, June/July 2006
Proceedings

Springer

Ian Gorton George T. Heineman
Ivica Crnkovic Heinz W. Schmidt
Judith A. Stafford Clemens A. Szyperski
Kurt Wallnau (Eds.)

Component-Based Software Engineering

9th International Symposium, CBSE 2006
Västerås, Sweden, June 29 – July 1, 2006
Proceedings

Springer

Volume Editors

Ian Gorton
National ICT Australia, Eveleigh, NSW 1430, Australia
E-mail: ian.gorton@nicta.com.au

George T. Heineman
WPI, Worcester, MA 01609, USA
E-mail: heineman@cs.wpi.edu

Ivica Crnkovic
Mälardalen University, 721 23 Västerås, Sweden
E-mail: ivica.crnkovic@mdh.se

Heinz W. Schmidt
Monash University, Clayton VIC 3800 , Australia
E-mail: heinz.schmidt@csse.monash.edu.au

Judith A. Stafford
Tufts University, Medford, MA 02155, USA
E-mail: jas@cs.tufts.edu

Clemens A. Szyperski
Microsoft Corp., Redmond, WA 98053, USA
E-mail: cszypers@microsoft.com

Kurt Wallnau
Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA
E-mail: kcw@sei.cmu.edu

Library of Congress Control Number: 2006927704

CR Subject Classification (1998): D.2, D.1.5, D.3, F.3.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-35628-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-35628-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11783565 06/3142 5 4 3 2 1 0

Preface

On behalf of the Organizing Committee I am pleased to present the proceedings of the 2006 Symposium on Component-Based Software Engineering (CBSE). CBSE is concerned with the development of software-intensive systems from reusable parts (components), the development of reusable parts, and system maintenance and improvement by means of component replacement and customization. CBSE 2006 was the ninth in a series of events that promote a science and technology foundation for achieving predictable quality in software systems through the use of software component technology and its associated software engineering practices.

We were fortunate to have a dedicated Program Committee comprising 27 internationally recognized researchers and industrial practitioners. We received 77 submissions and each paper was reviewed by at least three Program Committee members (four for papers with an author on the Program Committee). The entire reviewing process was supported by Microsoft's CMT technology. In total, 22 submissions were accepted as full papers and 9 submissions were accepted as short papers.

This was the first time CBSE was not held as a co-located event at ICSE. Hence special thanks are due to Ivica Crnkovic for hosting the event. We also wish to thank the ACM Special Interest Group on Software Engineering (SIGSOFT) for their sponsorship of CBSE 2005. The proceedings you now hold were published by Springer and we are grateful for their support. Finally, we must thank the many authors who contributed the high-quality papers contained within these proceedings. As the international community of CBSE researchers and practitioners continues to prosper, we expect the CBSE Symposium series to similarly attract widespread interest and participation.

May 2006

Ian Gorton

Organization

CBSE 2006 was sponsored by the Association for Computing Machinery (ACM) Special Interest Group in Software (SIGSOFT).

Organizing Committee

Program Chair: Ian Gorton (NICTA, Australia)
Steering Committee: Ivica Crnkovic
(Mälardalen University, Sweden)
George T. Heineman
(WPI, USA)
Heinz W. Schmidt
(Monash University, Australia)
Judith A. Stafford (Tufts University, USA)
Clemens Szyperski (Microsoft Research, USA)
Kurt Wallnau
(Software Engineering Institute, USA)

Program Committee

Uwe Assmann, Dresden University of Technology, Germany
Mike Barnett, Microsoft Research, USA
Judith Bishop, University of Pretoria, South Africa
Jan Bosch, Nokia Research Center, Finland
Michel Chaudron, University of Eindhoven, The Netherlands
Shiping Chen, CSIRO, Australia
Susan Eisenbach, Imperial College, UK
Dimitra Giannakopoulou, RIACS/NASA Ames, USA
Lars Grunske, University of Queensland, Australia
Richard Hall, LSR-IMAG, France
Dick Hamlet, Portland State University, USA
George Heineman, Worcester Polytechnic Institute, USA
Tom Henzinger, EPFL, Switzerland and UC Berkeley, USA
Paola Inverardi, University of L'Aquila, Italy
Jean-Marc Jezequel, IRISA (INRIA & Univ. Rennes 1), France
Bengt Jonsson, Uppsala University, Sweden
Dean Kuo, University of Manchester, UK
Magnus Larsson, ABB, Sweden
Kung-Kiu Lau, University of Manchester, UK
Nenad Medvidovic, University of Southern California, USA
Rob van Ommering, Philips, The Netherlands
Otto Preiss, ABB Switzerland

Ralf Reussner, University of Oldenburg, Germany
Douglas Schmidt, Vanderbilt University, USA
Jean-Guy Schneider, Swinburne University of Tech., Australia
Dave Wile, Teknowledge, Corp., USA
Wolfgang Weck, Independent Software Architect, Switzerland

Previous CBSE Workshops and Symposia

8th International Symposium on CBSE, Lecture Notes in Computer Science, Vol. 3489, Heineman, G.T. et al (Eds.), Springer, St. Loius, USA (2005)

7th International Symposium on CBSE, Lecture Notes in Computer Science, Vol. 3054, Crnkovic, I.; Stafford, J.A.; Schmidt, H.W.; Wallnau, K. (Eds.), Springer, Edinburgh, UK (2004)

6th ICSE Workshop on CBSE: Automated Reasoning and Prediction
http://www.sei.cmu.edu/pacc/CBSE6. Portland, Oregon (2003)

5th ICSE Workshop on CBSE: Benchmarks for Predictable Assembly
http://www.sei.cmu.edu/pacc/CBSE5. Orlando, Florida (2002)

4th ICSE Workshop on CBSE: Component Certification and System Prediction. Software Engineering Notes, 26(10), November 2001. ACM SIGSOFT Author(s): Crnkovic, I.; Schmidt, H.; Stafford, J.; Wallnau, K. (Eds.)
http://www.sei.cmu.edu/pacc/CBSE4-Proceedings.html. Toronto, Canada, (2001)

Third ICSE Workshop on CBSE: Reflection in Practice
http://www.sei.cmu.edu/pacc/cbse2000. Limerick, Ireland (2000)

Second ICSE Workshop on CBSE: Developing a Handbook for CBSE
http://www.sei.cmu.edu/cbs/icse99. Los Angeles, California (1999)

First Workshop on CBSE
http://www.sei.cmu.edu/pacc/icse98. Tokyo, Japan (1998)

Table of Contents

Full Papers

Short Papers

Defining and Checking Deployment Contracts for Software Components

Kung-Kiu Lau and Vladyslav Ukis

School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu, vukis}@cs.man.ac.uk

Abstract. Ideally in the deployment phase, components should be composable, and their composition checked. Current component models fall short of this ideal. Most models do not allow composition in the deployment phase. Moreover, current models use only deployment descriptors as deployment contracts. These descriptors are not ideal contracts. For one thing, they are only for specific containers, rather than arbitrary execution environments. In any case, they are checked only at runtime, not deployment time. In this paper we present an approach to component deployment which not only defines better deployment contracts but also checks them in the deployment phase.

1 Introduction

Component deployment is the process of getting components ready for execution in a target system. Components are therefore in binary form at this stage. Ideally these binaries should be composable, so that an arbitrary assembly can be built to implement the target system. Furthermore, the composition of the assembly should be checked so that any conflicts between the components, and any conflicts between them and the intended execution environment for the system, can be detected and repaired before runtime. This ideal is of course the aim of CBSE, that is to assemble third-party binaries into executable systems. To realise this ideal, component models should provide composition operators at deployment time, as well as a means for defining suitable deployment contracts and checking them.

Current component models fall short of this ideal. Most models only allow composition of components in source code. Only two component models, JavaBeans [7] and the .NET component model [6, 20], support composition of binaries. Moreover, current models use only deployment descriptors as deployment contracts [1]. These descriptors are not ideal contracts. They do not express contracts for component composition. They are contracts for specific containers, rather than arbitrary execution environments. In any case, they are checked only at runtime, not deployment time.

Checking deployment contracts at deployment time is advantageous because they establish component composability, and thus avoid runtime conflicts. Moreover, they also allow the assembly to be changed if necessary before runtime. Furthermore, conflicts due to incompatibilities between components and the target execution environment of the system into which they are deployed can be discovered before runtime.

In this paper we present an approach to component deployment which not only defines better contracts but also checks them in the deployment phase. It is based on a

I. Gorton et al. (Eds.): CBSE 2006, LNCS 4063, pp. 1–16, 2006.

pool of metadata we have developed, which components can draw on to specify their runtime dependencies and behaviour.

2 Component Deployment

We begin by defining what we mean by component deployment. First, we define a 'software component' along the lines of Szyperski [24] and Heinemann and Councill [10], viz. 'a software entity with contractual interfaces and contextual dependencies, defined in a component model'.[1]

Our definition of component deployment is set in the context of the component lifecycle. This cycle consists of three phases: *design*, *deployment* and *runtime* (Fig. 1).

Fig. 1. Software component lifecycle

In the design phase, a component is designed and implemented in source code, by a *component developer*. For example, to develop an Enterprise JavaBean (EJB) [18] component in the design phase, the source code of the bean is created in Java, possibly using an IDE like Eclipse. A component in this phase is not intended to run in any particular system. Rather, it is meant to be reusable for many systems.

In the deployment phase, a component is a binary, ready to be deployed into an application by a *system developer*. For example, in the deployment phase, an EJB is a binary ".class" file compiled from a Java class defined for the bean in the design phase.

For deployment, a component needs to have a deployment contract which specifies how the component will interact with other components and with the target execution environment. For example, in EJB, on deployment, a deployment descriptor describing the bean has to be created and archived with the ".class" file, producing a ".jar" file, which has to be submitted to an EJB container.

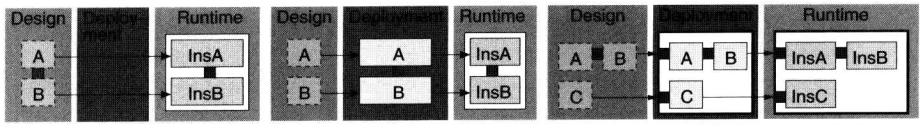
An important characteristic of the deployment phase is that the system developer who deploys a component may not be the same person as the component developer.

In the runtime phase, a component instance is created from the binary component and the instantiated component runs in a system. Some component models use containers for component instantiation, e.g. EJB and CCM [19]. For example, an EJB in binary form as a ".class" file archived in a ".jar" file in the deployment phase gets instantiated and is managed by an EJB container in the runtime phase.

2.1 Current Component Models

Of the major current software component models, only two, viz. JavaBeans and the .NET component model, allow composition in the deployment phase. To show this, we first relate our definition of the phases of the component lifecycle (Fig. 1) to current component models.

[1] Note that we deal with components obeying a component model and not with COTS [2].



Category 1: EJB, COM, CCM, Koala, SOFA, KobrA, ADLs, UML2.0, PECOS, Pin, Fractal

Category 2: JavaBeans

Category 3: .NET Component Model

Fig. 2. Current component models

Current component models can be classified according to the phases in which component composition is possible. We can identify three categories [16] as shown in Fig. 2.

In the first category, composition (denoted by the small linking box) happens only at design time. The majority of current models, viz. EJB, COM [3], CCM,ADLs (architecture description languages) [22],[2] etc. fall into this category. For instance, in EJB, the composition is done by direct method calls between beans at design time. An assembly done at design time cannot be changed at deployment time, and gets instantiated at runtime into executable instances (denoted by InsA, InsB.)

In the second category, composition happens only at deployment time. There is only one model in this category, viz. JavaBeans. In JavaBeans, Java classes for beans are designed independently at design time. At deployment time, binary components (".class" files) are assembled by the BeanBox, which also serves as the runtime environment for the assembly. Java beans communicate by exchanging events. The assembly is done at deployment time by the BeanBox, by generating and compiling an event adapter class.

In the third category, composition can happen at both design and deployment time. The sole member of this category is the .NET component model. In this model, components can be composed as in Category 1 at design time, i.e. by direct method calls. In addition, at deployment time, components can also be composed as in Category 2. This is done by using a container class, shown as a rectanglular box with a bold border. The container class hosts the binary components (".dll" files) and can make direct method calls into them.

Finally, current component models target either the desktop or the web environment, except for the .NET component model, which is unified for both environments. Having a component model that allows components to be deployed into both desktop and web environments enhances the applicability of the component model.

2.2 Composition in the Deployment Phase

Composition in the deployment phase can potentially lead to faster system development than design time composition, since binary components are bought from component suppliers and composed using (ideally pre-existing) composition operators, which can even be done without source code development. However, composition at component deployment time poses new challenges not addressed by current component models. These stem mainly from the fact that in the design phase, component developers design

[2] In C2 [17] new components can be added to an assembly at deployment time since C2 components can broadcast events; but new events can only be defined at design time.

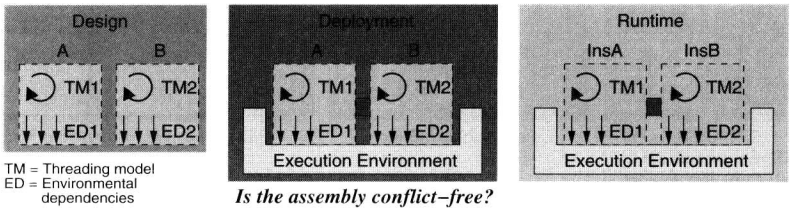


Fig. 3. Composition in deployment phase

and build components (in source code) independently. In particular, for a component, they may (i) choose any *threading model*; and (ii) define *dependencies on the execution environment*. This is illustrated by Fig. 3.

A component may create a thread inside it, use some thread synchronisation mechanisms to protect some data from concurrent access, or not use any synchronisation mechanisms on the assumption that it will not be deployed into an environment with concurrency.

Also each component supplier may use some mechanisms inside a component that require some resources from the system execution environment, thus defining the component's environmental dependencies. For instance, if a component uses socket communication, then it requires a network from the execution environment. If a component uses a file, then it requires file system access. Note that component suppliers do not know what execution environments their components will be deployed into.

In the deployment phase, the system developer knows the system he is going to build and the properties of the execution environment for the system. However, he needs to know whether any assembly he builds will be conflict-free (Fig. 3), i.e. whether (i) the threading models in the components are compatible; (ii) their environmental dependencies are compatible; (iii) their threading models and environmental dependencies are compatible with the execution environment; and (iv) their emergent assembly-specific properties are compatible with the properties of the execution environment if components are to be composed using a composition operator. The system developer needs to know all this before the runtime phase. If problems are discovered at runtime, the system developer will not be able to change the system. By contrast, if incompatibilities are found at deployment time, the assembly can still be changed by exchanging components.

By the execution environment we mean either the *desktop* or the *web* environment, and not a container (if any) for components. These two environments are the most widespread, and differ in the management of *system transient state* and *concurrency*. Since the component developer does not know whether the components will be deployed on a desktop or a web server, the system developer has to check whether the components and their assembly are suitable to run in the target execution environment.

2.3 Deployment Contracts

Deployment contracts express dependencies between components, and between them and the execution environment. As shown in [1], in most current component models a deployment contract is simply the interface of a component. In EJB and CCM,

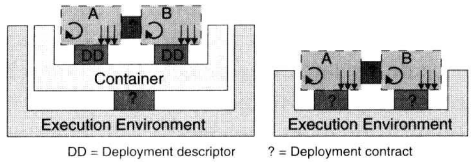


Fig. 4. Deployment contracts

deployment contracts are deployment and component descriptors respectively. As shown in Fig. 4, a deployment (or component) descriptor contractualises the management of a component by a container. However, the information about components inside the descriptors is not used to check whether components are compatible. Nor is it used to check whether a component can be deployed in an execution environment.

By contrast, our approach aims to check conflicts between components; and, in the presence of a component container, between the container and the execution environment; in the absence of a container, between components and the execution environment. This is illustrated by Fig. 4, where the question marks denote our deployment contracts, in the presence or absence of containers.

We can also check our deployment contracts, so our approach addresses the challenge of deployment time composition better than existing component models that allow deployment time composition, viz. the. NET component model and JavaBeans. In the .NET component model, *no* checking for component compatibilities is done during deployment. In JavaBeans, the BeanBox into which beans are deployed, is deployed on the desktop environment, and it checks whether beans can be composed together by checking whether events emitted by a source bean can be consumed by the target bean, by matching event source with event sink. However, this check is not adequate with regard to threading models and environment dependencies, as shown by the following example.

Example 1. Consider a Java bean that creates a thread inside itself to perform some long-running task in the background and sends an event to another bean from within that thread. The target bean may have problems. For example, if the target bean makes use of a COM component that requires a single-threaded apartment, and the bean is invoked from different threads, the component assembly is bound to fail.

This shows that the threading model of the source bean, namely sending an event from an internally created thread, and the environmental dependency of the target bean, namely the use of the COM component requiring a single-threaded apartment, are incompatible. The assembly will fail at runtime even though the BeanBox's check for component (event) compatibility is passed.

3 Defining Deployment Contracts

In this section we discuss how we define suitable deployment contracts. Our approach is based on metadata about component environmental dependencies and threading models. To determine and create suitable metadata, we studied the two most comprehensive, operating system-independent frameworks [9] for component development: J2EE [23]

and .NET Framework [25]. In particular, we studied the core APIs of these two frameworks in order to identify where and how a component can incur environmental dependencies and influences on its threading model. The comprehensiveness and wide application of these frameworks should imply the same for the metada we create. We define deployment contracts using these metadata[3] as attributes that the component developer is obliged to attach to components he develops.

3.1 Environmental Dependencies

A component incurs an environmental dependency whenever it makes use of a resource offered by the operating system or the framework using which it is implemented. For each resource found this way we created an attribute expressing the semantics of the environmental dependency found. Each attribute has defined parameters and is therefore parameterisable. Moreover, each attribute has defined *attribute targets* from the set {component, method, method's parameter, method's return value, property}. An *attribute target* defines the element of a component it can be applied to.

To enable a developer to express resource usage as precisely as possible, we allow each attribute to have (a subset of) the following parameters: 1) 'UsageMode': {Create, Read, Write, Delete} to indicate the usage of the resource. Arbitrary combinations of values in this set are allowed. However, here we assume that inside a component, creation, if specified, is always done first. Also, deletion, if specified, is always done last; 2) 'Existence': {Checked, Unchecked} to indicate whether the component checks for existence of a resource or makes use of it assuming it is there; 3) 'Location': {Local, Remote} to indicate whether a resource required by component is local on the machine the component is deployed to or is remote; 4) 'UsageNecessity': {Mandatory, Optional} to indicate whether a component will fail to execute or will be able to fulfil its task if the required resource is not available.

Meaningful combinations of the values of these parameters allow an attribute to appear in different forms (120 for an attribute with all 4 parameters) which have to be analysed differently.

In addition to these four parameters, any attribute may have other parameters specific to a particular environmental dependency. For instance, consider an attribute on a component's method expressing an environmental dependency to a COM component shown in Fig. 5. (Such a component was used in Example 1.) The component has a method "Method2" that has the attribute "UsedCOMComponent" attached. The attribute has (1) shows the COM GUID used by the component; (2) says that three parameters:

```
public class B
{
 [UsedCOMComponent("DC577003–3436–470c–8161–EA9204B11EBF",      (1)
  COMAppartmentModel.Singlethreaded,                            (2)
  UsageNecessity.Mandatory)]                                    (3)
 public void Method2(...) {...}
}
```

Fig. 5. A component with an environmental dependency

[3] A full list and details can be found in [14].

Table 1. Categories of resource usage and component developer's obligations

1	*Usage of an operating-system resource.* For instance: Files, Directories, Input/Ouput Devices like Printers, Event Logs, Performance Counters, Processes, Residential Services, Communication Ports and Sockets.
2	*Usage of a resource offered by a framework.* For instance: Application and Session State storages offered by J2EE and .NET for web development, Communication Channels to communicate with remote objects.
3	*Usage of a local resource.* For instance: Databases, Message Queues and Directory Services.
4	*Usage of a remote resource.* For instance: Web Services or Web Servers, Remote Hosts, and resources from Category 3 installed remotely.
5	*Usage of a framework.* For instance: DirectX or OpenGL.
6	*Usage of a component from a component model.* For instance: a Java Bean using a COM component via EZ JCOM [8] framework.

the used COM component requires a single-threaded environment; (3) says that the usage of the COM component is mandatory. Furthermore, implicitly the attribute says that the component requires access to a file system as well as Windows Registry since COM components have to be registered there with GUID.

We have analysed the pool of attributes we have created, and as a result we can define categories of resource usage for which the component developer is obliged to attach the relevant attributes to their component's elements. The categories are shown in Table 1:

Using binary components with relevant attributes from the categories in Table 1, it is possible at deployment time to detect potential conflicts based on contentious use of resources from Table 1.

Finally, metadata about environmental dependencies can be used to check for mutual compatibility of components in an assembly. For instance, if a component from an assembly requires continuous access to a file in the file system in the write mode but another component in the assembly also writes to the same file but creates it afresh without checking whether it has existed before, the first component may lose its data and the component assembly may fail to execute.

3.2 Threading Models

A component can create a thread, register a callback, invoke a callback on a thread [4, 5], create an asynchronous method [11], make use of thread-specific storage [21] or access a resource requiring thread-affine access,[4] etc. For each of these cases, we created an attribute of the kind described in Section 3.1 expressing the semantics of the case.

For instance, consider an attribute expressing the creation of a thread by a component shown in Fig. 6. (Such a component was used in Example 1.) The component has a method "Method1" that has the attribute "SpawnThread" attached. The parameter (1) indicates the number of threads spawned. If this method is composed with another component's method requiring thread affinity, the composition is going to fail.

[4] Thread-affine access to a resource means that the resource is only allowed to be accessed from one and the same thread.

```
public class A
{
 [SpawnThread(1)]                    (1)
 public void Method1(...) {...}
}
```

Fig. 6. A component with a defined threading model

Table 2. Categories of threading issues and component developer's obligations

1	*Existence of an asynchronous method.*
2	*Registration or/and invocation of a callback method.*
3	*Existence of reentrant or/and thread-safe methods.*
4	*Existence of component elements requiring thread-affine access.*
5	*Existence of Singletons or static variables.*
6	*Spawning a thread.*
7	*Usage of Thread-specific storage.*
8	*Taking as a method parameter of returning a synchronisation primitive.*

We have analysed the pool of attributes we have created, and as a result we can define categories of threading issues for which the component developer is obliged to attach the relevant attributes to their components. These categories are shown in Table 2:

Using binary components with attributes from the categories shown in Table 2, it is possible at component deployment time to detect potential conflicts based on inappropriate usage of threads and synchronisation primitives by components in an assembly. It is also possible to point out potential deadlocks in a component assembly.

In total, for both environmental dependencies and threading models, we have created a pool of about 100 metadata attributes[5]. Now we show an example of their use.

Example 2. Consider Example 1 again.The two incompatible Java beans are shown in Fig. 7 with metadata attributes from Sections 3.1 and 3.2. Using these attributes we can detect the incompatibility of the beans at deployment time.[6]

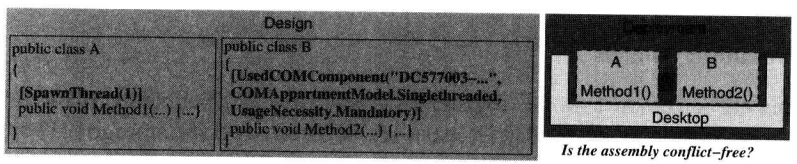


Fig. 7. Example 1 using metadata attributes

In the design phase, The two beans are the ones in Figs. 5 and 6. In the deployment phase, by performing an analysis of the metadata attributes attached to the components, we can deduce that method "A.Method1()" invokes the method "B.Method2()"

[5] In .NET Framework v2.0 there are about 200 attributes, but they are only checked at runtime.

[6] Note that this problem may also arise in other component models.