MANCHESTER 1824

The University of Manchester

Computer Science
University of Manchester

A Reasoning Framework for Deployment Contracts Analysis

Kung-Kiu Lau and Vladyslav Ukis

Department of Computer Science
University of Manchester
Preprint Series
CSPP-37

A Reasoning Framework for Deployment Contracts Analysis

Kung-Kiu Lau and Vladyslav Ukis

June 2006

Abstract

Most current component models regard component's interface to be component's deployment contract [1]. If components are designed independently and composed in deployment phase, viz. in binary form, the interfaces of components are checked for compatibility to establish component composition. However, a component developer may choose arbitrary threading model for and make use of any resources in execution environment of a component. In order to analyse whether threading models of components and their environmental dependencies in a component assembly deployed in an execution environment are mutually compatible, means for expressing component's threading models and environmental dependencies are necessary. In [10, 11] we proposed a set of parameterisable metadata, as deployment contracts for components, for that purpose. In this report we present a reasoning framework for analysing the deployment contracts. Using the framework, it is possible on component deployment to spot conflicts in component assemblies deployed into an execution environment due to incompatible threading models of components as well as their environmental dependencies.

Keywords: components, deployment contracts for components, component composition, compositional reasoning

Copyright © 2000, University of Manchester. All rights reserved. Reproduction (electronically or by other means) of all or part of this work is permitted for educational or research purposes only, on condition that no commercial gain is involved.

Recent preprints issued by the Department of Computer Science, Manchester University, are available on WWW via URL `http://www.cs.man.ac.uk/preprints/index.html` or by ftp from `ftp.cs.man.ac.uk` in the directory `pub/preprints`.

Contents

1 Introduction 6

2 Detectable Conflicts using Component Deployment Contracts 6
2.1 Conflicts due to absence of resources in the execution environment required by components in an assembly . . . 8
2.2 Conflicts due to contentious use of available resources by components in an assembly 9
2.3 Conflicts due to incompatible threading models of components in an assembly . . 11
2.4 Conflicts due to incompatible threading model of a component and concurrency management of the execution environment . . . 12
2.5 Conflicts due to incompatible state model of a component and state management of the execution environment . . . 14

3 Reasoning Framework for Deployment Contracts Analysis 15

4 Deployment Contracts Analyser 26
4.1 Support for Component Connectors . . . 31
4.1.1 Support for Automated Component Composition . . . 32

5 Examples of Deployment Contracts Analysis 33
5.1 Spotting conflicts due to absence of resources in the execution environment required by components in an assembly . . . 33
5.1.1 Example 1 . . . 33
5.2 Spotting conflicts due to contentious use of available resources by components in an assembly . . . 34
5.2.1 Example 2 . . . 34
5.2.2 Example 3 . . . 36
5.2.3 Example 4 . . . 37
5.3 Spotting conflicts due to incompatible threading models of components in an assembly . . . 38
5.3.1 Example 5 . . . 38
5.3.2 Example 6 . . . 40
5.3.3 Example 7 . . . 41
5.4 Spotting conflicts due to incompatible threading model of a component and concurrency management of the execution environment . . . 42
5.4.1 Example 8 . . . 42
5.4.2 Example 9 . . . 43
5.4.3 Example 10 . . . 44
5.5 Spotting conflicts due to incompatible state model of a component and state management of the execution environment . . . 44
5.5.1 Example 11 . . . 44
5.6 Spotting combined conflicts . . . 46
5.6.1 Example 12 . . . 46
5.6.2 Example 13 . . . 47
5.6.3 Example 14 . . . 48
5.6.4 Example 15 . . . 50

6 Evaluation **52**

7 Conclusion **53**

8 Appendix **53**

8.1 Code outline for checking mutual compatibility of deployment contracts of components with respect to usage of resources in execution environment 53

8.2 Code outline for checking mutual compatibility of deployment contracts of components with respect to their threading models in consideration of state and concurrency management of execution environment 56

List of Figures

1 Composition in deployment phase. 7
2 Deployment contracts. 7
3 Conflicts due to absence of resources in the execution environment required by components in an assembly. 9
4 Conflicts due to contentious use of available resources by components in an assembly. 9
5 Conflicts due to contentious use of a file by components in an assembly. 10
6 Conflicts due to different use of a database connection by components in an assembly. 11
7 Conflicts due to incompatible threading models of components in an assembly. . . 11
8 Conflicts due to incompatible use of thread-specific storage by components in an assembly. 12
9 Conflicts due to incompatible threading model of a component and concurrency management in web execution environment. 13
10 Conflicts due to incompatible threading model of a component and concurrency management in web execution environment. 14
11 Conflicts due to incompatible state model of a component and state management of web execution environment. 15
12 Overview of Deployment Contracts Analyser. 26
13 View of Deployment Contract of Component. 27
14 Creating a simulation of a component assembly. 28
15 Defining Assembly's Execution Environment. 29
16 Deployment Contracts Analysis. 30
17 Loading Component Connectors. 31
18 Generating Composition Plan. 32
19 Example 1. 33
20 Deployment contracts analysis for the Example 1. 34
21 Example 2. 34
22 Deployment contracts analysis for the Example 2. 35
23 Example 3. 36
24 Deployment contracts analysis for the Example 3. 37
25 Example 4. 38
26 Deployment contracts analysis for the Example 4. 38
27 Example 5. 39
28 Deployment contracts analysis for the Example 5. 39
29 Example 6. 40
30 Deployment contracts analysis for the Example 6. 40
31 Example 7. 41
32 Deployment contracts analysis for the Example 7. 41
33 Example 8. 42
34 Deployment contracts analysis for the Example 8. 42
35 Example 9. 43
36 Deployment contracts analysis for the Example 9. 43
37 Example 10. 44
38 Deployment contracts analysis for the Example 10. 44

39 Example 11. 45
40 Deployment contracts analysis for the Example 11. 45
41 Example 12. 46
42 Deployment contracts analysis for the Example 12. 47
43 Example 13. 48
44 Deployment contracts analysis for the Example 13. 48
45 Example 14. 49
46 Deployment contracts analysis for the Example 14. 49
47 Example 15. 50
48 Deployment contracts analysis for the Example 15. 51

List of Tables

1 Concurrency management in the web execution environment. 13
2 State management in the web execution environment. 14
3 Subsequent usage of a resource RR by two components **C1** and **C2** without considering RR's state. 18
4 For a single component: RR's UsageMode with Creation without Deletion vs. Existence 19
5 For a single component: RR's UsageMode with Deletion without Creation vs. Existence 20
6 For a single component: RR's UsageMode with Creation and Deletion vs. Existence 21
7 For a single component: RR's UsageMode without Creation and Deletion vs. Existence 21
8 RR's state transition chart 22
9 Desktop environment's properties vs. assembly-specific properties 23
10 System instantiation modes in the web environment vs. assembly-specific properties 25

1 Introduction

Most current component models support composition only in design phase [14]. Only two component models, JavaBeans [4] and the .NET component model [3, 17], support composition in deployment phase, i.e. when components are binaries. These component models regard component's interface to be component's deployment contract [1]. When components are designed independently and composed in deployment phase the interfaces of components are checked for compatibility to establish component composition. However, a component developer may choose arbitrary threading model for and make use of resources in execution environment of a component. In order to analyse whether threading models of components and their environmental dependencies in a component assembly deployed in an execution environment are mutually compatible, means for expressing component's threading models and environmental dependencies are necessary. In [11] we proposed a set of parameterisable metadata, as deployment contracts for components, for that purpose. The set of metadata is general purpose and comprehensive since it is created by analysis the two most comprehensive, operating system-independent frameworks [5] for component development: J2EE [20, 15] and .NET Framework [22, 16]. In this report we present a reasoning framework for analysing the deployment contracts. Using the framework, it is possible on component deployment to spot conflicts in component assemblies deployed into an execution environment due to incompatible threading models of components as well as their environmental dependencies.

The report is organised as follows. We begin in Section 2 by presenting conflicts that can be detected using our deployment contracts from [11]. In Section 3 we present an algorithm that can detect the conflicts having components accompanied with the metadata. Subsequently, in Section 4 we show a tool – Deployment Contracts Analyser – implementing the algorithm. Furthermore, in Section 5 we present examples of component assemblies where we can spot problems shown in Section 2 by applying the algorithm from Section 3. Finally, we evaluate our approach in Section 6 and conclude in Section 7.

2 Detectable Conflicts using Component Deployment Contracts

Component Deployment Contracts are used to check compositionality of components when they are composed 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 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. 1.

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. 1), 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.

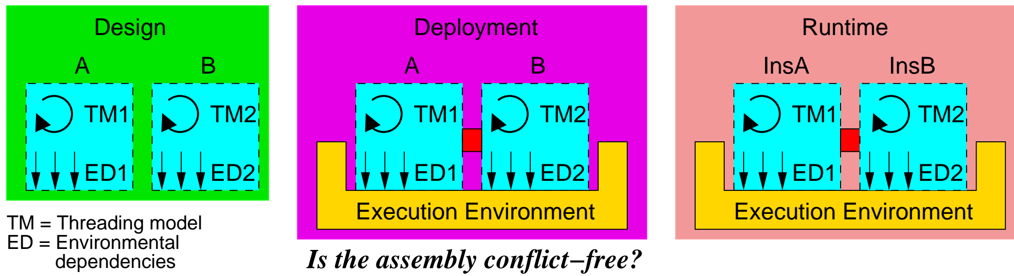


Figure 1: Composition in deployment phase.

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. Therefore, the aim of our deployment contracts is to enable reasoning about components' incompatibilities at deployment time.

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.

As shown in Fig. 2, a deployment (or component) descriptor contractualises the management of a component by a container.

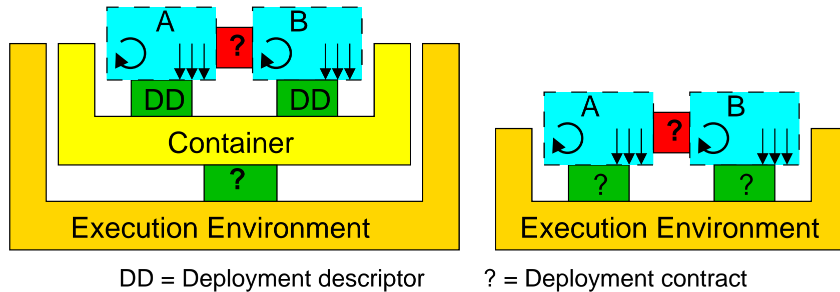


Figure 2: Deployment contracts.

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 resolve 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. 2, 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 but components can potentially be deployed in both desktop and web environment. 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, no checking of beans' threading models and environmental dependencies is performed.

Our deployment contracts are represented by metadata in a metadata pool that can be divided in two main categories:

- Metadata expressing component's environmental dependencies and
- Metadata expressing component's threading model.

Following these categories of metadata, detectable conflicts in component assemblies deployed into an execution environment are due to:

1. Absence of resources in the execution environment that are required by components in an assembly.
2. Contentious use of available resources by components in the assembly.
3. Incompatible threading models of components in the assembly.
4. Incompatible threading model of a component from the assembly and concurrency management of the execution environment.
5. Incompatible state model of a component and state management of the execution environment.

In the following sections, we elaborate on each kind of conflicts above.

2.1 Conflicts due to absence of resources in the execution environment required by components in an assembly

Conflicts due to the absence of resources in the execution environment required by components in an assembly arise when at design time component developers make use of resources, which are not available in the the execution environment the assembly is deployed to. This is illustrated in Figure 3.

Component A is designed with use of resources R1 and R2. For instance, R1 can be a database and R2 a web service. Component B is designed with use of resources R3 and R4. For instance, R3 can be a socket and R4 a file.

The components A and B are deployed into an execution environment where only the resources $R1'$, corresponding to R1, and $R4'$, corresponding to R4, are available. Assuming that R1 is a database required by the component A, $R1'$ represents the database in the execution environment. Moreover, assuming that R4 is a file, $R4'$ represents the file system in the execution environment.

The resources $R2'$, corresponding to R2, and $R3'$, corresponding to R3, required by components A and B respectively are not available in the execution environment of the assembly AB. Assuming that R2 is a web service, $R2'$ represents network on the local machine. Furthermore, assuming that R3 is a socket, $R3'$ also represents network. Thus, network is not available in the

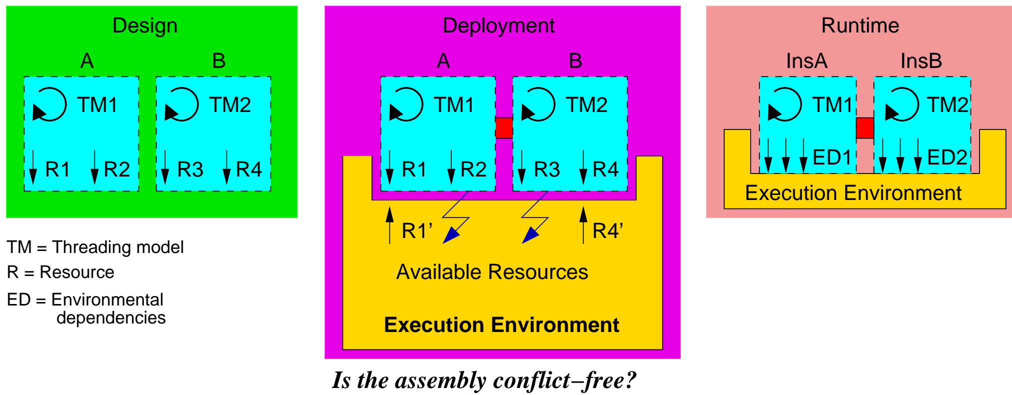


Figure 3: Conflicts due to absence of resources in the execution environment required by components in an assembly.

execution environment the assembly AB is deployed to. Since both components require network access, component A by using a web service (R2) and component B by using a socket (R3), neither of the two components is suitable to run in the execution environment. The assembly AB is bound to fail at runtime.

Deployment Contracts allow the component developer to flexibly specify the necessity of a resource's use. If a component can fulfill its task in the absence of a resource, the component developer can specify in the component's deployment contract that the use of the resource is optional. For instance, if the component A from Figure ?? would not necessarily require the resource R2, but would make use of it only if it is available, the component would be able to run in the execution environment.

2.2 Conflicts due to contentious use of available resources by components in an assembly

Conflicts due to contentious use of available resources by components in an assembly arise when at design time several component developers make use of the same resource in incompatible ways. This is illustrated in Figure 4.

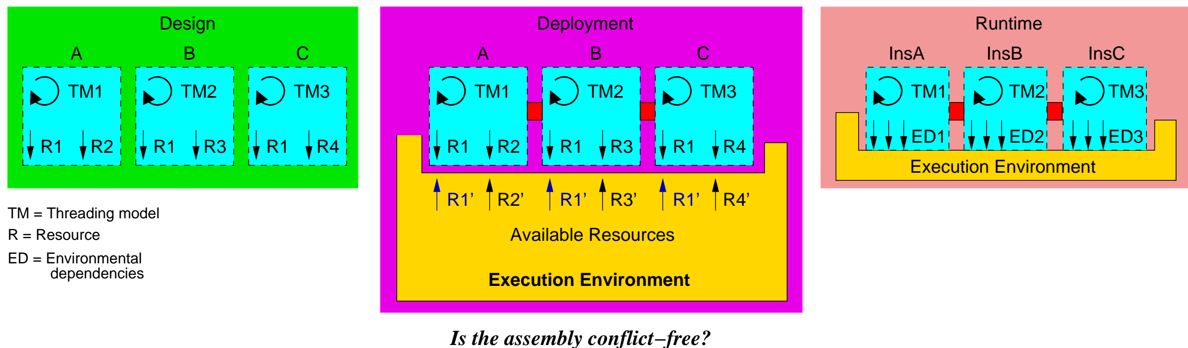


Figure 4: Conflicts due to contentious use of available resources by components in an assembly.

Components A, B and C are designed independently. All of them make use of a resource R1.

For instance, R1 can be a file. At deployment time, an assembly ABC is created using some composition operators. The resource $R1'$, file system, is available in the assembly's execution environment. The arrangement of components in the assembly ABC suggests that the resource R1 is first accessed by the component A, second by the component B and third by the component C.

R1 can be created, deleted, written to and read from by a component. Moreover, a component can check (or not) the resource R1 for existence before using it. These parameters are known about a resource through the deployment contract of component making use of it.

For instance, assuming R1 is a file and components A, B and C from the assembly ABC in Figure 4 use it in the following ways: component A creates and writes to the file; component B reads from the file and deletes it; and component C reads from the file as well shown in Figure 5.

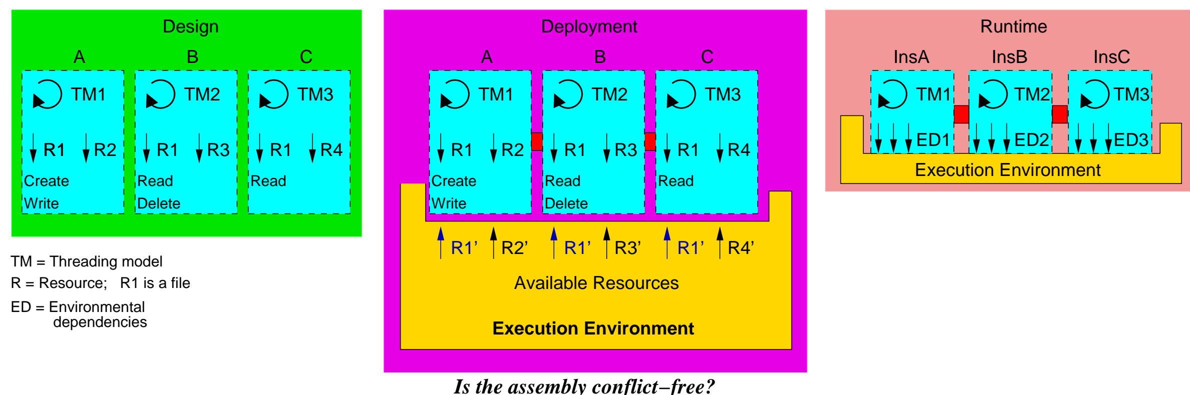


Figure 5: Conflicts due to contentious use of a file by components in an assembly.

The file is created by the component A. Subsequently, the component A writes to the file. After that, the component B reads from the file as well as deletes it. The file does not exist after its use by the component B. However, the component C tries to read from the file and will fail, unless it is optional for the component. The assembly ABC is bound to fail at runtime.

Other contentious combinations of used resources based on their creation and deletion by the components in an assembly are possible. They can be detected by analysing Deployment Contracts of components.

Other conflicts of the kind shown in Figure 4 can arise if a resource is used (just) differently by components in the assembly. Assuming that a resource is a database connection, a problematic assembly is shown in Figure 6.

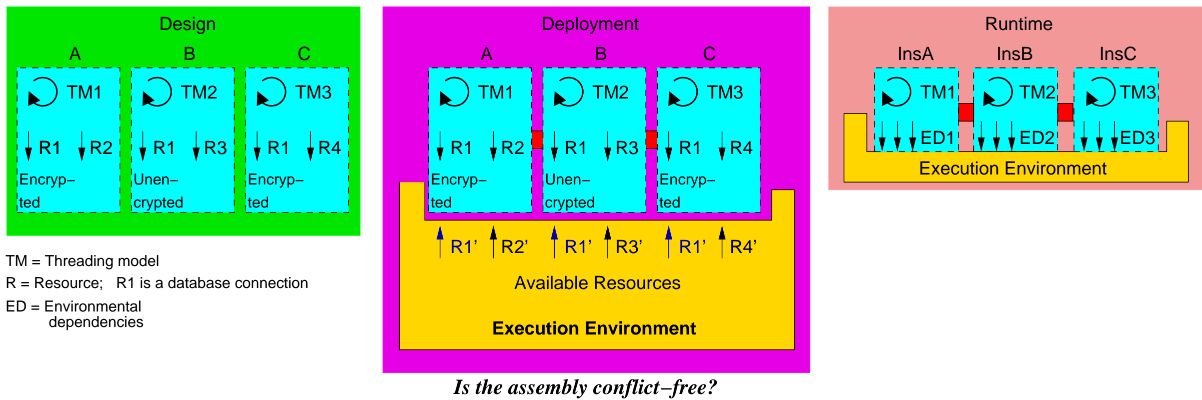


Figure 6: Conflicts due to different use of a database connection by components in an assembly.

Components A, B and C in Figure 6 are designed with use of a database connection. At component deployment time, an assembly ABC is created. Assuming that all components in the assembly connect to the same database, we have an assembly where the component A uses an encrypted database connection, the component B an unencrypted one, and the component C an encrypted one. Depending on the system developer's security requirements, the assembly ABC can be an acceptable one or not. On the one hand, if the system is required to be secure, the assembly ABC is unacceptable and the component B has to be replaced by another one which uses an encrypted database connection. On the other hand, if the system is not required to be secure, the assembly ABC may well be acceptable.

An analysis of Deployment Contracts of components can spot such potential conflicts and point them out to the system developer.

2.3 Conflicts due to incompatible threading models of components in an assembly

Conflicts due to incompatible threading models of components in an assembly arise if a component creates a thread inside and invokes another component on that thread. Depending on the threading model of the latter component, threading issues can arise. This is illustrated in Figure 7.

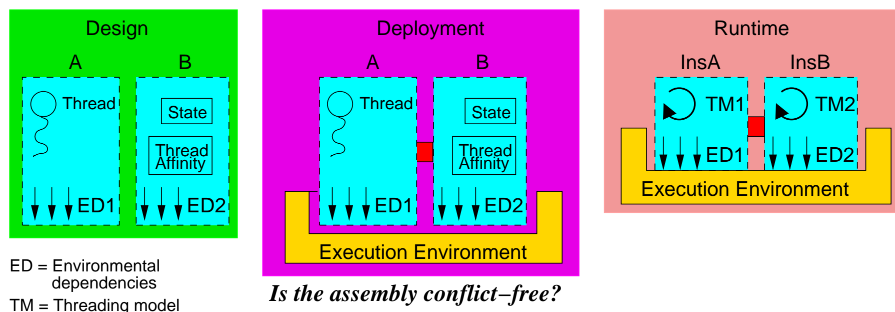


Figure 7: Conflicts due to incompatible threading models of components in an assembly.

Component A (Figure 7) is designed with a thread inside. Component B is designed with state

inside as well as some parts which require thread affinity [1].

At deployment time, an assembly AB is created using a composition operator. The composition operator composes the components A and B in a way that component A invokes component B on a thread that is created inside A. Each time component A is called, it creates a thread and invokes component B on it. Therefore, if component A is called multiple times, multiple threads can execute concurrently in component B. Since component B has some state inside which is unprotected from concurrent access by multiple threads using a thread synchronisation primitive, state corruption will occur in component B. Furthermore, since component B has some parts that require thread-affine access but are accessed by multiple threads in the assembly AB, the component B is going to fail. The assembly AB is bound to fail at runtime.

Another kind of problems with components' threading models lies in incompatible use of thread-specific storage [18] (TSS). Figure 8 shows two components A and B that are designed with use of TSS.

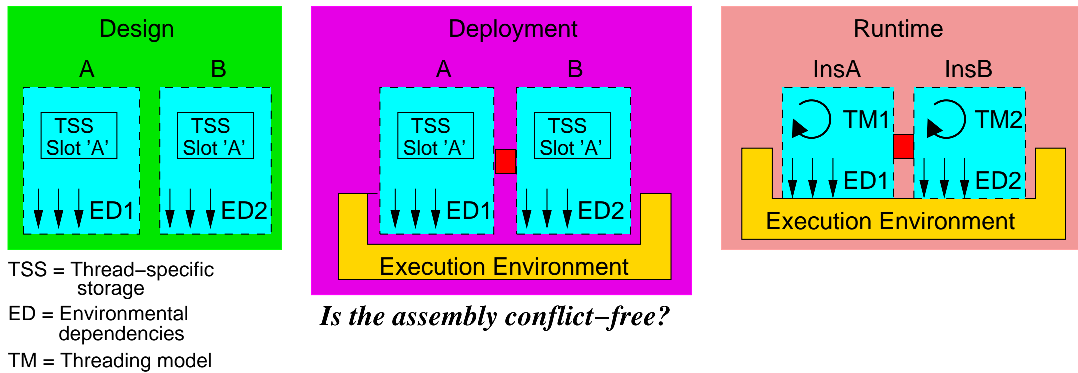


Figure 8: Conflicts due to incompatible use of thread-specific storage by components in an assembly.

Component A uses a TSS Slot with the name "A", component "B" uses the same TSS Slot. If both components write different information to and read from the slot, the components are going to fail since each component will find information written by another component in the slot. Therefore, the assembly AB is bound to fail at runtime.

Other conflicts with components' threading models exist. Deployment Contracts of components can express threading models of components. They can be analysed to spot incompatible threading models.

2.4 Conflicts due to incompatible threading model of a component and concurrency management of the execution environment

Conflicts due to incompatible threading model of a component from an assembly and concurrency management of the execution environment arise in the web execution environment. In the web execution environment there are different assembly instantiation modes. Depending on the assembly instantiation mode, concurrency management in the web environment can be defined as shown in Table 1.

[1]Thread-affinity access to some code means that the code is only allowed to be accessed from one and the same thread.

Assembly Instantiation Mode	Assembly instance per request	Assembly instance per user (browser) session	Assembly instance for all requests	Pool of synchronised assembly instances for all requests
Concurrency Management	An assembly instance is accessed by one thread	An assembly instance can be accessed by multiple threads *sequentially*	Concurrent access of an assembly instance by multiple threads	An assembly instance can be accessed by multiple threads *sequentially*

Table 1: Concurrency management in the web execution environment.

As shown in Table 1 if the assembly instantiation mode is 'assembly instance per request', the assembly is accessed by one thread. If the assembly instantiation mode is either 'assembly instance per user session' or 'pool of synchronised assembly instances for all requests' the main thread affinity is not guaranteed and each request may be issued by another thread. If the assembly instantiation mode is 'assembly instance for all requests', the assembly is accessed concurrently by multiple threads.

Figure 9 demonstrates two components A and B. Component A is designed with state inside. Component B is designed with some parts requiring thread affinity.

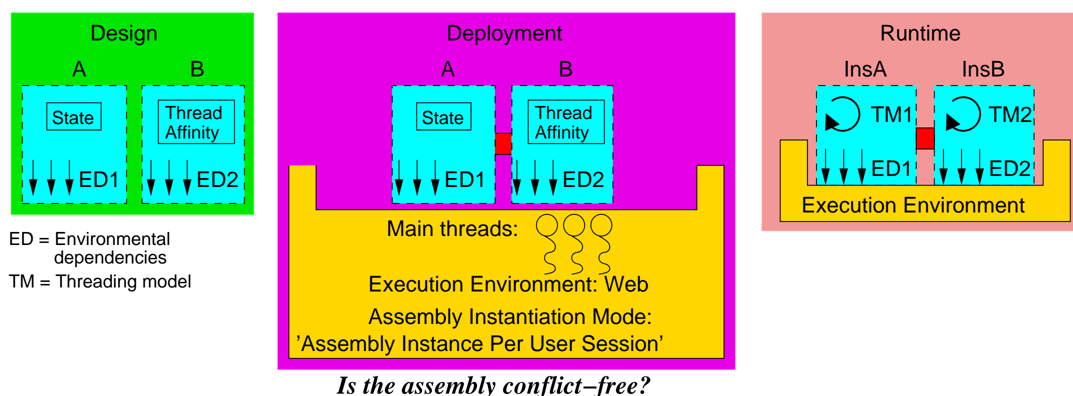


Figure 9: Conflicts due to incompatible threading model of a component and concurrency management in web execution environment.

At deployment time the components A and B are composed to form an assembly AB. The assembly AB is deployed in the web execution environment with assembly instantiation mode 'assembly instance per user session'. In such environment, main thread affinity is not guaranteed. Component A has state inside. Since different main threads do not access the state concurrently, the component will not fail in the execution environment. Component B requires thread affine access to some of its parts. Since there is no thread affinity of the main thread in the execution environment, the component B is not suitable to run there. The component B has to be replaced by another one since the assembly AB is bound to fail at runtime.

Figure 10 shows the same assembly AB but deployed into the web environment with assembly instantiation mode 'assembly instance for all requests'.

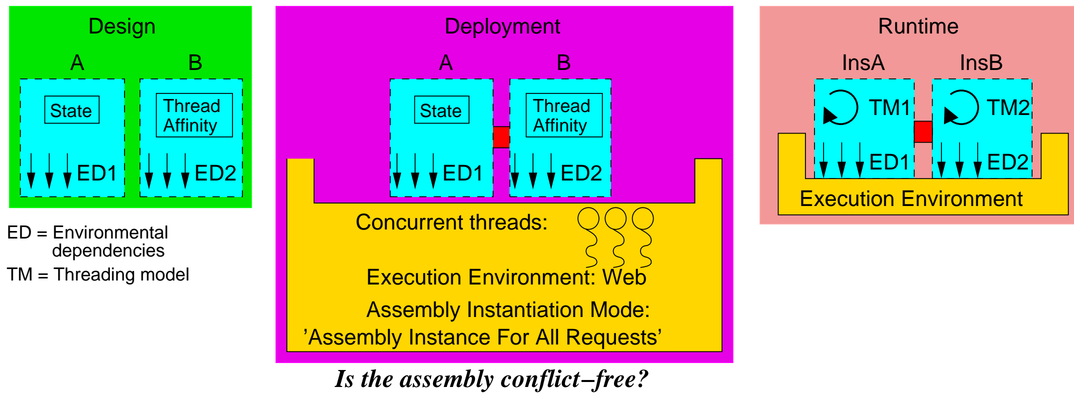


Figure 10: Conflicts due to incompatible threading model of a component and concurrency management in web execution environment.

In this execution environment, the assembly is accessed concurrently by multiple threads. Component A has some state inside which is unprotected from concurrent access by multiple threads. Therefore, it is unsuitable to run in this environment. Component B requires thread affinity. Since the assembly will be concurrently accessed by multiple threads in the execution environment, component B is unsuitable as well. Therefore, the assembly AB is bound to fail at runtime.

Note that the assembly AB will run smoothly in the desktop environment because it guarantees main thread affinity and does not impose multiple threads on the assembly.

2.5 Conflicts due to incompatible state model of a component and state management of the execution environment

Conflicts due to incompatible state model of a component in an assembly and state management of the execution environment arise in the web execution environment. In the web execution environment there are different assembly instantiation modes. Depending on the assembly instantiation mode, state management of components can be defined as shown in Table 2.

Assembly Instantiation Mode	Assembly instance per request	Assembly instance per user (browser) session	Assembly instance for all requests	Pool of synchronised assembly instances for all requests
Assembly transient state Management	Assembly state is not retained among requests	Assembly state is retained during a user session	Assembly state is retained among all requests	Assembly state is not retained among requests

Table 2: State management in the web execution environment.

As shown in Table 2 if the assembly instantiation mode is ‘assembly instance per request’ or ‘pool of synchronised assembly instances for all requests’, the assembly state is not retained among requests to the assembly. If the assembly instantiation mode is either ‘assembly instance per user session’, assembly state is retained during a user session, but not among user sessions. If the assembly instantiation mode is ‘assembly instance for all requests’, assembly state is retained among all requests to the assembly.

Figure 11 demonstrates two components A and B. Component A is designed with state inside. Component B is also designed with state, which is stored in a state storage[2].

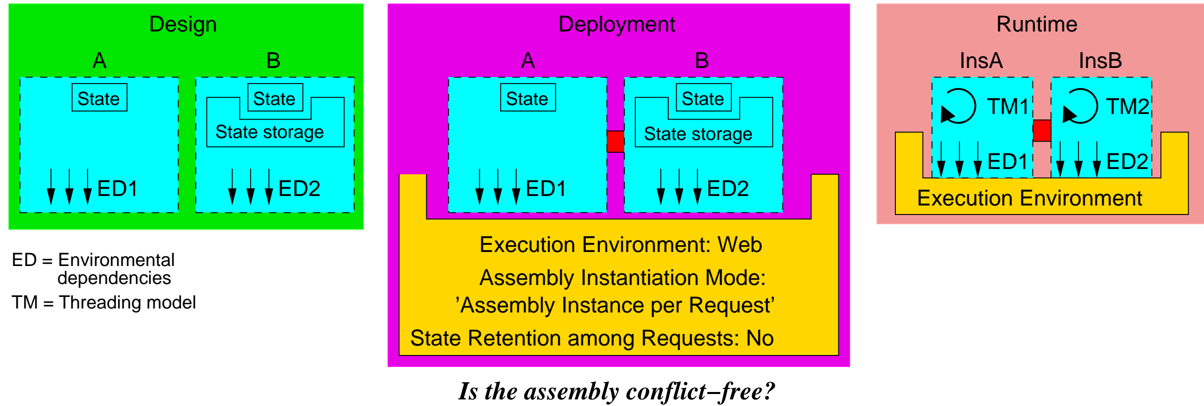


Figure 11: Conflicts due to incompatible state model of a component and state management of web execution environment.

At deployment time the components A and B are composed to form an assembly AB. The assembly AB is deployed in the web execution environment with assembly instantiation mode 'assembly instance per request'. In such environment, state is not retained among requests to the assembly. Component A has state inside. Since state is not retained among requests to the assembly, the component will lose its state after each request. Therefore, component A is not suitable to run in the execution environment. Component B has also state inside. The state is stored in state storage offered by the execution environment. The data in the state storage is retained for a time period depending on the kind of state storage used. (There are two kinds of state storage: application and session state storage. Data in the application state storage is retained for the lifetime of the application. Data in the session state storage is retained for a user session.) Therefore, although the execution environment does not retain state among requests to the assembly, state of component B is safely stored in a state storage. Thus, component B can run in the execution environment.

The component A from the assembly AB has to be replaced by another one since the assembly is bound to fail at runtime.

Having identified conflicts that can be detected using Deployment Contracts of components [11], in the next section we present a reasoning framework using which the conflicts can be discovered.

3 Reasoning Framework for Deployment Contracts Analysis

The Reasoning Framework for Deployment Contracts Analysis is based on the algorithm presented in this section. To retrieve Deployment Contract of a component, reflection is used throughout the algorithm.

Input to the Algorithm:

[2] In the web environment, there are two kinds of state storage: application and session state storage. For further information on these state storages, the reader is referred to [11], where they are explained in context of Deployment Contracts for components.

1. Components with Deployment Contracts.

2. Defined execution environment – Desktop or Web environment and Resources available in the environment.

3. Component connections – Definition which methods are connected, in which order; and for a connection of two methods whether parameter piping is done, i.e. return parameter of a method is fed as an input parameter into the next method. Furthermore, an indication whether a connection is recurrent.

Output of the Algorithm:

A set of ERRORS, WARNINGS and HINTS with the meaning:

1. ERROR – Component has to be replaced by another one.

2. WARNING – Possible error. System developer's knowledge is required to decide whether the Warning means an error for their assembly or can be ignored.

3. HINT – A minor finding, which may result in a warning or even error for a particular assembly. The system-developer has to decide whether a finding is harmless for their assembly.

The Deployment Contracts Analysis Algorithm steps:

1. *Analysis of Mutual Compatibility of Deployment Contracts of Components in the Assembly with Respect to Usage of Resources in the Assembly's Execution Environment* [3]

   (a) If resources available in the assembly's execution environment are defined:

   (b) For each component:

   (c) For each component-level, method-level and property-level attribute of the component:

   (d) If the attribute represents a resource that requires some resource(s) in the execution environment (for instance a socket represents a resource that requires Network from the execution environment):

   (e) Which resource(s) (**R1**) in the execution environment is required by the resource represented by the attribute? For instance: if the attribute represents a file, R1 is File System; if the attribute represents a socket, R1 is Network; if the attribute represents a web service, R1 is Network too.

   (f) If the attribute has the parameter Location and it is set to Remote, then on the machine the component is deployed to Network is required but R1 is required remotely.

      i. If Network is available in the execution environment, a HINT is issued that R1 is required remotely.

      ii. If Network is not available

[3] In pseudocode. A code outline of most important parts can be found in Appendix 8.1.

A. If the attribute does not have the parameter UsageNecessity, or the parameter UsageNecessity exists and is set to Mandatory, an ERROR is issued since the required resource is not available in the execution environment.

B. If the parameter UsageNecessity exists and is set to Optional, a HINT is issued stating that although the required resource is not available in the execution environment, the component will be able to fulfil its task since the use of the resource is optional for the component.

(g) If the attribute does not have Location parameter, or the Location parameter exists and is set to Local

i. If R1 is not available in the execution environment, either step 1(f)iiA or step 1(f)iiB is done.

(h) If no ERROR is issued in steps 1a – 1g, i.e. all requirements of components with respect to resources can be satisfied by the execution environment:

(i) If component connections are specified:

(j) For each connection we consider *how the resource represented* **(RR)** by an attribute is used by the components in the connection. (For instance: if a remote file (RR) is required by several components in the connection, the components require Network (R1) from the execution environment on the machine they are deployed to. If Network is available in the execution environment, we consider *how* the file (RR) is used by the components.)

i. For all components in the connection:

ii. If a component method or property has the attribute "RequirePreviousMethodInvocation" or "RequirePreviousPropertyInvocation" attached, check whether the previous method is called on the component before the method call as required by the attribute. If not, issue an ERROR.

iii. If an attribute on a component method or property representing usage of a resource is found, find the next component in the connection using the same attribute either at method or property or component level.

A. If parameters of the two attributes on the components show that the same resource RR is used, follow the algorithm below.
Abbreviations: UsageMode.Create = C, UsageMode.Read = R, UsageMode.Write = W, UsageMode.D = D.
There are 15 meaningful combinations of the above values a component can use: D, W, R, C, W&D, R&D, C&D, W&R, C&W, C&R, R&W&D, C&R&D, C&W&D, C&R&W, C&W&R&D.
The state of a represented resource RR shows whether the RR is created (viz. exists) or deleted (viz. does not exist) or it is unknown if it has been created or deleted: {Created, Deleted, Unknown}

- Set RR's state to Unknown
- Locate C1.M1 in Table 3.

C1.M1 ⟶ C2.M2 means **C1.M1()** is directly or indirectly via other components connected to and thus is invoked prior to **C2.M2()**

C1.M1 ⟶ C2.M2: **C2.M2** → **C1.M1** ↓	C, C&W, C&R, C&R&W: (Table 4) C&D, C&R&D, C&W&D, C&R&W&D: (Table 6)	W&D, R&D, R&W&D (Table 5)	W, R&W: (Table 7)	D: (Table 5)	R: (Table 7)
D (Table 5)	–	use of not existing RR ***(UNE RR)***	UNE RR	deletion of deleted RR	UNE RR
W (Table 7)	data loss possible ***(DLP)***	DLP, HINT	HINT	del. of del. RR	–
R (Table 7)	DLP	HINT(except R&D)	HINT	–	–
C (Table 4)	–	–	–	–	–
W&D (Table 5)	–	UNE RR	UNE RR	del. of del. RR	UNE RR
R&D (Table 5)	–	UNE RR	UNE RR	del. of del. RR	UNE RR
C&D (Table 6)	–	UNE RR	UNE RR	del. of del. RR	UNE RR
R&W (Table 7)	DLP	DLP, HINT	HINT	del. of del. RR	HINT
C&W (Table 4)	DLP	DLP, HINT	HINT	del. of del. RR	HINT
C&R (Table 4)	DLP	HINT(except R&D)	HINT	–	–
R&W&D (Table 5)	–	UNE RR	UNE RR	del. of del. RR	UNE RR
C&R&D (Table 6)	–	UNE RR	UNE RR	del. of del. RR	UNE RR
C&W&D (Table 6)	–	UNE RR	UNE RR	del. of del. RR	UNE RR
C&R&W (Table 4)	DLP	DLP, HINT	HINT	del. of del. RR	–
C&R&W&D (Table 6)	–	UNE RR	UNE RR	del. of del. RR	UNE RR

HINT means that in the presence of multiple threads operating concurrently in C1.M1 and C2.M2 Readers and Writers Problem [21] may occur. The system developer is referred to a concurrent programming language to check this kind of problem, if required.

Table 3: Subsequent usage of a resource RR by two components **C1** and **C2** without considering RR's state.

- Look up what happens in C1.M1 using the corresponding Table shown in brackets and the current state of RR.
  The Tables in brackets check the relation of the parameter UsageMode and Existence for an RR by a single component. The Tables are built according to the combinations of values of parameter UsageMode divided into 4 categories:
  - RR usage with Creation but without Deletion: C, C&W, C&R, C&R&W (Table 4). Here we assume that in combined usages Creation is done before Read and Write

	RR Usage Modes: **Create, Create&Write, Create&Read, Create&Read&Write**
	Component creates RR itself but does not delete it.
	The component does not delete RR which may be undesirable depending on RR's type – HINT if the component is the last one in the chain.
Existence. Checked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate* ***A and B****. Otherwise perform* ***either A or B****:* **A.** If RR exists, it is not created – OK. In cases C&W, C&R, C&R&W, RR is used – OK. If RR cannot be used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, it is created – OK. If RR cannot be created or used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT.
Existence. Unchecked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate* ***A and B****. Otherwise perform* ***either A or B****:* **A.** If RR exists, which is unchecked, it is created afresh. This may cause problems depending on RR – WARNING. In cases C&W, C&R, C&R&W, RR is used – OK. If RR cannot be created or used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, it is created – OK. If RR cannot be created or used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT.

Table 4: For a single component: RR's UsageMode with Creation without Deletion vs. Existence

- RR usage with Deletion but without Creation: D, W&D, R&D, R&W&D (Table 5). Here we assume that Read and Write is done before Deletion

	RR Usage Modes: **Delete, Write&Delete, Read&Delete, Read&Write&Delete**
	Component does not create RR itself but deletes it.
	The component assumes that RR exists. RR either must be there or must be created by another component before use by the component.
Existence. Checked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate **A and B**.* **A.** If RR exists, it is used and deleted in cases W&D, R&D, R&W&D, or just deleted in case D – OK. If RR cannot be used or deleted: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is checked, it cannot be used or deleted. If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. *If known whether RR is created (=exists) or deleted (= does not exist) perform **either A or B**.* **A.** If RR exists, it is used and deleted in cases W&D, R&D, R&W&D, or just deleted in case D – OK. If RR cannot be used or deleted: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is checked, it cannot be used or deleted. If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – ERROR.
Existence. Unchecked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate **A and B**.* **A.** If RR exists, it is used and deleted in cases W&D, R&D, R&W&D, or just deleted in case D – OK. If RR cannot be used or deleted: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is unchecked, it cannot be used or deleted. If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – WARNING. *If known whether RR is created (=exists) or deleted (= does not exist) perform **either A or B**.* **A.** If RR exists, it is used and deleted in cases W&D, R&D, R&W&D, or just deleted in case D – OK. If RR cannot be used or deleted: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is unchecked, it cannot be used or deleted. If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – ERROR (possibly abnormal situation).

Table 5: For a single component: RR's UsageMode with Deletion without Creation vs. Existence

- RR usage with Creation and Deletion: C&D, C&R&D, C&W&D, C&W&R&D (Table 6). Here we assume that Creation is done first, Deletion is done last, and Read and Write is done between Creation and Deletion

	RR Usage Modes: **Create&Delete, Create&Read&Delete, Create&Write&Delete, Create&Read&Write&Delete**
	Component creates and deletes RR itself.
Existence. Checked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate **A and B**. Otherwise perform **either A or B**:* **A.** If RR exists, it is not created, used and deleted – OK. If RR cannot be used or deleted: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is checked, it is created, used and deleted If RR cannot be created, used or deleted: If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT.
Existence. Unchecked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate **A and B**. Otherwise perform **either A or B**:* **A.** If RR exists, it is created afresh, used and deleted – creating RR afresh may cause problems depending on RR's type – WARNING. If RR cannot be created, used or deleted: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is unchecked, it is created, used and deleted. If RR cannot be created, used or deleted: If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT.

Table 6: For a single component: RR's UsageMode with Creation and Deletion vs. Existence

– RR usage without Creation and Deletion: R, W, R&W (Table 7)

	RR Usage Modes: **Read, Write, Read&Write**
	Component neither creates nor deletes RR itself.
	The component assumes that RR exists. RR either must be there or must be created by another component before use by the component. The component does not delete RR which may be undesirable depending on RR's type – HINT if the component is the last one in the chain.
Existence. Checked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate **A and B**.* **A.** If RR exists, it is used – OK. If RR cannot be used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is checked, it cannot be used: If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. *If known whether RR is created (=exists) or deleted (= does not exist) perform **either A or B**.* **A.** If RR exists, it is used – OK. If RR cannot be used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is checked, it cannot be used: If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – ERROR.
Existence. Unchecked	*If unknown whether RR is created (=exists) or deleted (= does not exist) indicate **A and B**.* **A.** If RR exists, it is used – OK. If RR cannot be used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is unchecked, it cannot be used: If RR cannot be used: If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – WARNING. *If known whether RR is created (=exists) or deleted (= does not exist) perform **either A or B**.* **A.** If RR exists, it is used – OK. If RR cannot be used: if UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – HINT. **B.** If RR does not exist, which is unchecked, it cannot be used: If RR cannot used: If UsageNecessity is Optional – OK, if UsageNecessity is Mandatory – ERROR (possibly abnormal situation).

Table 7: For a single component: RR's UsageMode without Creation and Deletion vs. Existence

- Change the current state of RR according to RR's UsageMode by C1.M1 using Table 8:

RR's Usage Mode by Component	RR's State
D	Changes to "Deleted"
W	Remains unchanged
R	Remains unchanged
C	Changes to "Created"
W&D	Changes to "Deleted"
R&D	Changes to "Deleted"
C&D	Changes to "Deleted"
R&W	Remains unchanged
C&W	Changes to "Created"
C&R	Changes to "Created"
R&W&D	Changes to "Deleted"
C&R&D	Changes to "Deleted"
C&W&D	Changes to "Deleted"
C&R&W	Changes to "Created"
C&R&W&D	Changes to "Deleted"

Table 8: RR's state transition chart

- Locate C2.M2 in Table 3
- Look up in Table 3 what can happen when going from C1.M1's usage mode of RR to C2.M2's
- Look up what happens in C2.M2 using the corresponding Table shown in brackets and the current state of RR
- Change the current state of RR according to RR's UsageMode by C2.M2 using Table 8

B. If the connection is marked as recurrent, perform the analysis in the previous step for the last and the first component in the connection as if they called each other.

iv. Perform the analysis in step 1(j)iii for component level attributes only.

v. If piping of values is used between two methods of two components, check whether the types of the values match. If not, issue a WARNING and show the types. If the type of the values is "object", check "UsedType" attribute. If the types do not match, issue a WARNING and show the types. If there is no "UsedType" attribute attached to either component, issue an ERROR saying that the deployment contract of component is not properly defined.

vi. If a synchronisation primitive is exchanged between methods of components – WARNING that this may be the cause of a potential deadlock.

vii. Specific parameters of each attribute are analysed and warnings and hints are issued. For instance, if components in a component connection use a database and all but one of the components uses unencrypted connection, a WARNING is issued.
If a cryptography certificate file is used it is hinted which cryptography algorithm has been used to create the certificate. If a communication channel is used, it is hinted which communication protocol for data transfer and which serialisation method for data serialisation is used. This information is used by the system developer to judge whether the component is suitable for their system.

2. *Analysis of Mutual Compatibility of Deployment Contracts of Components in the Assembly with Respect to their Threading Models in Consideration of State and Concurrency Management of Assembly's Execution Environment* [4]

   (a) If component connections are defined:

   (b) For each connection:

   (c) For each component in the connection:

   (d) Define whether the component is stateful/stateless and multithreaded/singlethreaded:

      i. A *component is stateful* if it has the attribute [AccessComponentTransientState] attached to any of its elements. Otherwise, it is *stateless*.

      ii. A *component is multithreaded* if it has at least one of the following attributes attached at any level: [SpawnThread], [AsynchronousMethod], [IssueCallback(ExecutingThread.InternallyCreatedThread)]. Otherwise it is *singlethreaded*.

   (e) Depending on whether the component is stateful/stateless and multithreaded/ singlethreaded, combine the values and decide whether it is one of the following: {"singlethreaded stateless", "singlethreaded stateful", "multithreaded stateless", "multithreaded stateful"}.

   (f) Decide whether the assembly is one of the following {"singlethreaded stateless", "singlethreaded stateful", "multithreaded stateless", "multithreaded stateful"}:

      i. If at least one of the components in the assembly is multithreaded the *assembly is multithreaded*. Otherwise it is *singlethreaded*.

      ii. If at least one of the components in the assembly is stateful, the *assembly is stateful*. Otherwise, it is *stateless*.

   Combine the values above and make the decision. Go on with checking whether the assembly is suitable for its execution environment in step 2g.

   (g) If the type of the Execution Environment is defined:

   (h) If Execution Environment Type is Desktop, see Table 9:

	Assembly in the desktop environment
System transient state management	Assembly state is retained among all requests to the system
Concurrency management	Main thread affinity is guaranteed
Singlethreaded, Stateless Assembly	OK
Singlethreaded, Stateful Assembly	OK
Multithreaded, Stateless Assembly	See step 2(h)i ignoring case with state.
Multithreaded, Stateful Assembly	See step 2(h)i.

Table 9: Desktop environment's properties vs. assembly-specific properties

[4] In pseudocode. A code outline of most important parts can be found in Appendix 8.2.

i. For each multithreaded component in the assembly:

ii. For each component in the connection chain after the multithreaded component:

iii. For each component-level, property-level, method-level, method input and return parameter-level attribute of the component:

iv. If the multithreaded component is represented by [IssueCallback] attribute, or [AsynchronousMethod] attribute; and the connection is recurrent (recurrent connection means in this situation that a request may be issued while the callback from previous request (or just previous request) is being executed thus creating a truly multithreaded scenario):

   A. If the attribute represents a component's element requiring thread-affine access – ERROR.

   B. If the attribute represents a component's element accessing component's transient state in not read-only mode: If the component element or any component element enclosing it is not marked as reentrant or thread-safe – ERROR.

   C. If the attribute represents a Singleton: If the component element or any component element enclosing it is not marked as reentrant or thread-safe – ERROR.

   D. If the attribute represents a static variable: If the component element or any component element enclosing it is not marked as reentrant or thread-safe – ERROR.

v. If the multithreaded component is represented only by [SpawnThread] attribute:

   A. If the attribute represents a component's element requiring thread-affine access – WARNING.

   B. If the attribute represents a component's element accessing component's transient state in not read-only mode: If the component element or any component element enclosing it is not marked as reentrant or thread-safe – WARNING.

   C. If the attribute represents a Singleton: If the component element or any component element enclosing it is not marked as reentrant or thread-safe – WARNING.

   D. If the attribute represents a static variable: If the component element or any component element enclosing it is not marked as reentrant or thread-safe – WARNING.

(i) If Execution Environment Type is Web, see Table 10:

	Assembly instance per request	Assembly instance per user (browser) session	Assembly instance for all concurrent requests	Pool of synchronised assembly instances for all concurrent requests
Assembly transient state Managm.	Assembly state is not retained among requests	Assembly state is retained during a user session	Assembly state is retained among all concurrent requests	Assembly state is not retained among concurrent requests
Concurrency Managm.	An assembly instance is accessed by one thread	An assembly instance can be accessed by multiple threads *sequentially*	Concurrent access of an assembly instance by multiple threads	An assembly instance can be accessed by multiple threads *sequentially*
Single-threaded, Stateless Assembly	OK	Main thread affinity not guaranteed – See step 2(h)ivA	Assembly becomes multithreaded – See steps 2(h)ivA,2(h)ivC, 2(h)ivD for all components	Main thread affinity not guaranteed – See step 2(h)ivA
Single-threaded, Stateful Assembly	State has to be specially retained by using application or session state storage. If neither is used – ERROR	Main thread affinity not guaranteed – See step 2(h)ivA. State is only retained for a user session – WARNING	Assembly becomes multithreaded – See steps 2(h)ivA, 2(h)ivB, 2(h)ivC,2(h)ivD for all components	Main thread affinity not guaranteed – See step 2(h)ivA. State has to be specially retained by using application or session state storage. If neither is used – ERROR
Multi-threaded, Stateless Assembly	See step 2(h)v ignoring case with state	See step 2(h)i ignoring case with state. Main thread affinity not guaranteed – See step 2(h)ivA for all comps.	See step 2(h)i ignoring case with state. Main thread affinity not guaranteed – 2(h)ivA for all c. Additional threads induced by environment – 2(h)i for non-recurrent connections	See step 2(h)i ignoring case with state. Main thread affinity not guaranteed – 2(h)ivA
Multi-threaded, Stateful Assembly	See step 2(h)v; State has to be specially retained by using application or session state storage. If neither is used – ERROR	See step 2(h)i. Main thread affinity not guaranteed – See step 2(h)ivA for all comps. State is only retained for a user session – WARNING	See step 2(h)i. Main thread affinity not guaranteed – 2(h)ivA for all c. Additional threads induced by environment – 2(h)i for non-recurrent connections	See step 2(h)i. Main thread affinity not guaranteed – 2(h)ivA State has to be specially retained by using application or session state storage. If neither is used – ERROR

Table 10: System instantiation modes in the web environment vs. assembly-specific properties

Having presented the reasoning framework for deployment contracts analysis that can detect conflicts in component assemblies introduced in Section 2, in the next section we present an implementation of it. The implementation automates the process of analysis of deployment contracts of components for component assemblies thus greatly enhancing the usefulness of the approach for system developers.

4 Deployment Contracts Analyser

The reasoning framework for deployment contracts analysis from Section 3 is the basis for the tool we have implemented, and named Deployment Contracts Analyser (DCA). The overall view of the tool is shown in Figure 12.

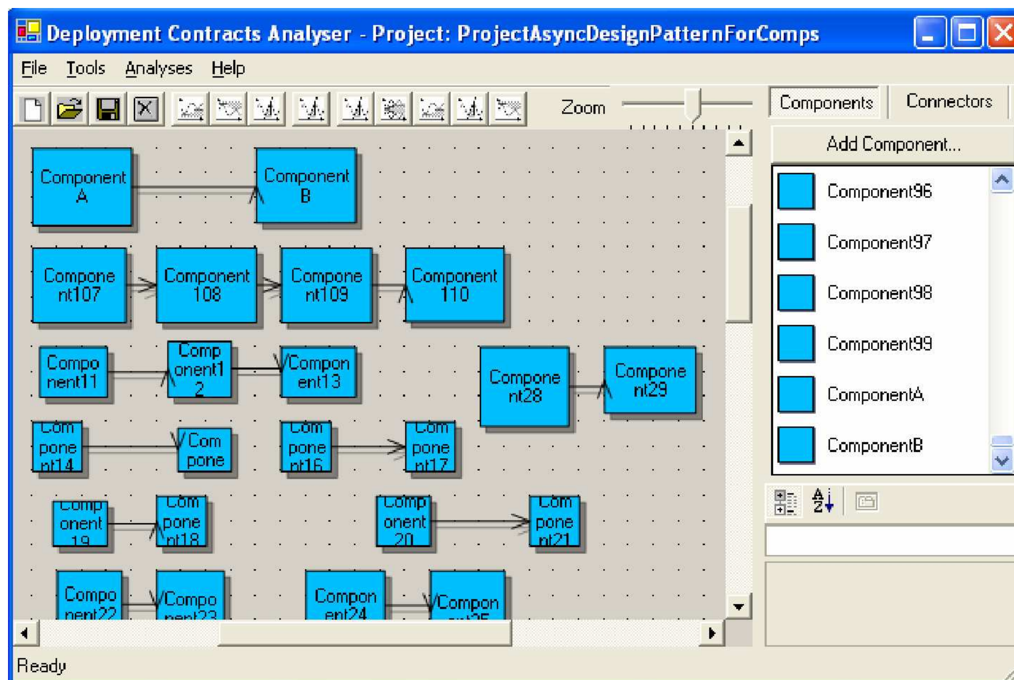


Figure 12: Overview of Deployment Contracts Analyser.

The Deployment Contracts Analyser takes as input binary components with deployment contracts. It allows simulation of assemblies of these components. Furthermore, it allows specification of an execution environment for an assembly. Finally, based on components with deployment contracts, a defined assembly of the components, and an execution environment for the assembly, the Deployment Contracts Analyser can automatically perform Deployment Contracts Analysis of Components shown in Section 3.

The Deployment Contracts Analyser follows the following workflow:

1. Independently designed binary components are loaded using a tab card shown on the right in Figure 12.

2. Loaded components are shown on the right hand side of the Deployment Contracts Analyser and can be dragged from the tab card and dropped on to the surface on the left (Figure 12). To view a component's deployment contract, the Deployment Contracts Analyser supports another view of the project shown in Figure 13.

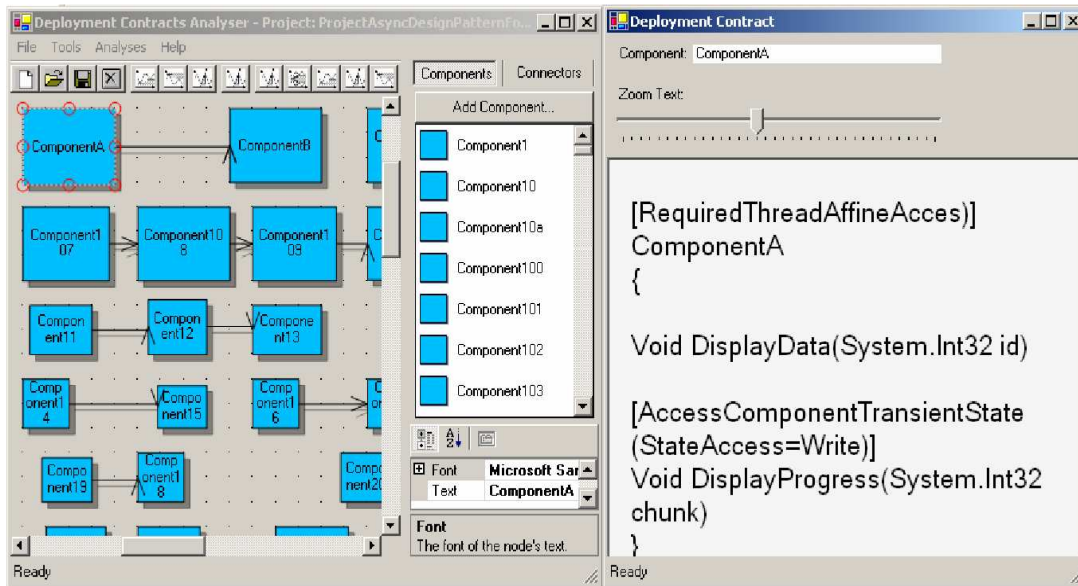


Figure 13: View of Deployment Contract of Component.

Figure 13 shows a selected component A in the left upper corner. Its deployment contract is shown on right hand side in Figure 13. Deployment contract of component is shown every time a component gets selected in this project view.

3. Two components on the surface can be visually connected using lines to indicate a component connection. To specify method connections of components on the surface and thereby to actually create a simulation of an assembly of components, the Deployment Contracts Analyser supports another view of the project shown in Figure 14.

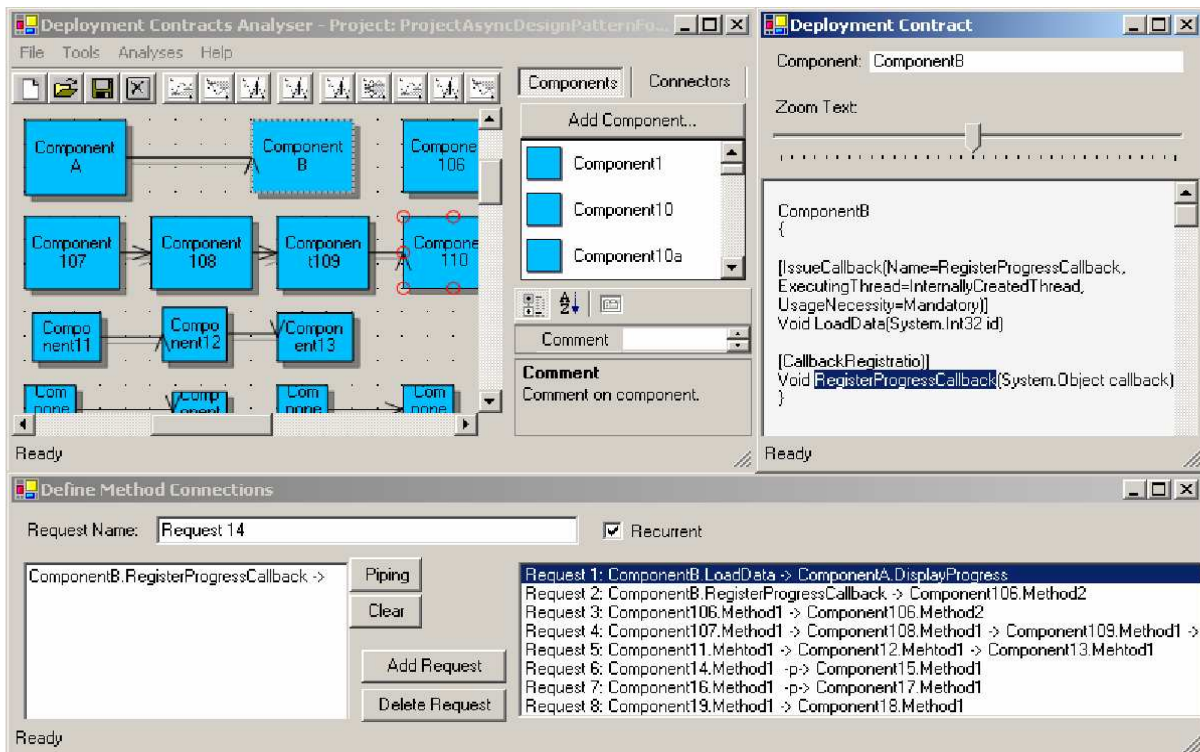


Figure 14: Creating a simulation of a component assembly.

To create a simulation of a component assembly, methods of components have to be connected. This is achieved by:

(a) Selecting a component on the surface on the left hand side in Figure 14. Selected component's methods are automatically shown on the right hand side in Figure 14.

(b) Selecting a method name of the component as shown on the right hand side in Figure 14. In the Figure, method 'RegisterProgressCallback' of the component B is selected. Selected method is automatically shown on the left in the lower part of Figure 14.

(c) In the lower part of Figure 14, the system developer can specify whether the return value of the selected method of component will be piped as a parameter into the next component in the connection. This is done by pressing the button 'Piping'. Furthermore, if the connection can be used more than once in the lifetime of the assembly, the system developer can specify this by checking the checkbox 'Recurrent'.

(d) To add the next component to the connection, steps 3a – 3c are repeated till all components needed are connected.

(e) Finally, the connection can be stored by pressing 'Add Request' button.

(f) To create other method connections, steps 3a – 3e are repeated. An assembly is completely specified when all method connections are defined.

4. The next step is to specify execution environment for the assembly defined in previous step. Deployment Contracts Analyser supports this with a GUI shown in Figure 15.

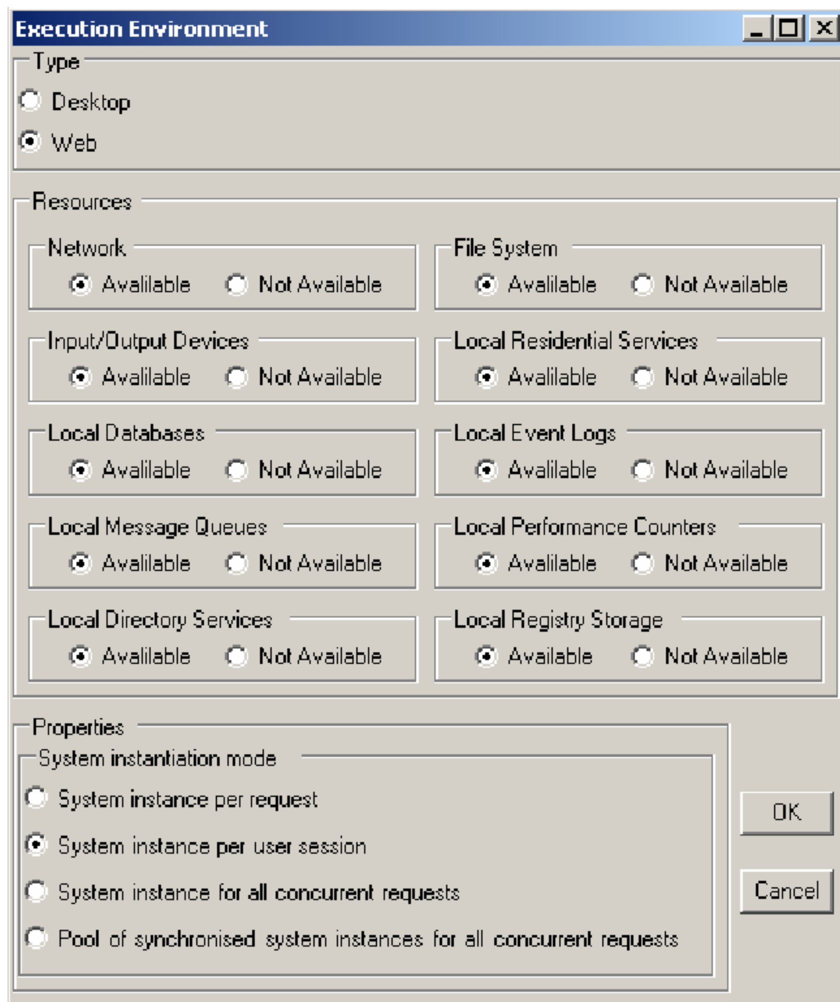


Figure 15: Defining Assembly's Execution Environment.

In the upper part of Figure 15 the type of the execution environment is specified: desktop or web environment. In the centre of Figure 15 resources, resource available in the execution environment can be defined. Finally, the lower part of Figure 15 is activated by the Deployment Contracts Analyser only if the type of the execution environment is web. It is used to specify assembly instantiation mode in the web environment.

5. Having defined an assembly of components as well as an execution environment for it, the Deployment Contracts Analyser can perform Deployment Contracts Analysis from Section 3. This is supported by the project view shown in Figure 16.

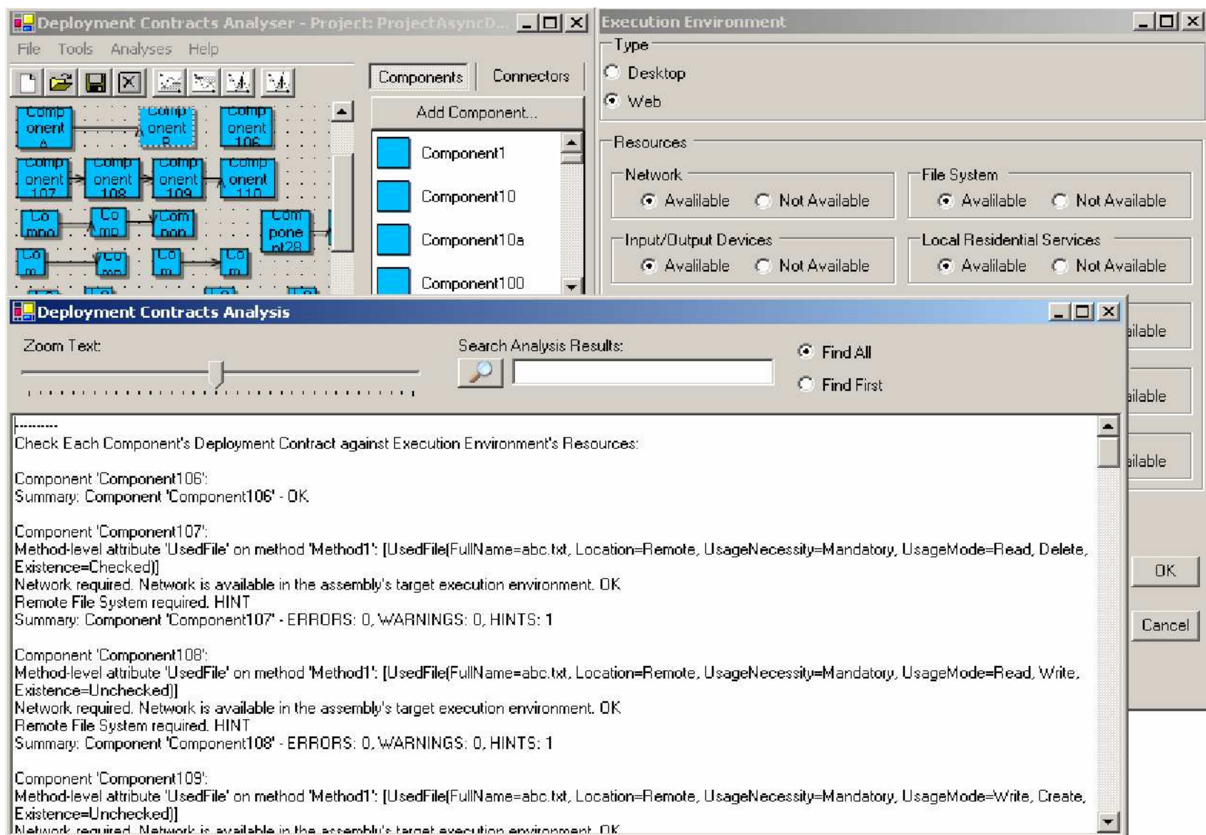


Figure 16: Deployment Contracts Analysis.

Results of deployment contracts analysis are shown in the lower part of Figure 16. In order to facilitate the system developer the task of browsing through the results, it is possible to search them by entering a word in the text box in the upper part of Figure 16 and pressing the button with the magnifying glass on it. Words matching the search criterion are highlighted making it easier for the system developer to find a specific issue in the analysis results.

6. Finally, if the system developer finds some issues in the results of deployment contracts analysis, they can either

   (a) change assembly's execution environment by repeating the steps 4 and 5 or

   (b) change method connections by repeating the steps 3 and 5 or

   (c) exchange components in the assembly by repeating the steps 1 – 3 and 5 or

   (d) perform a mixture of the above steps.

Once an assembly is found with acceptable results of deployment contracts analysis, it can safely execute at runtime.

4.1 Support for Component Connectors

Component connectors in current established component models supporting deployment time composition, viz. Java Beans and .NET Component Model, are not generic and self-contained. They have to be produced (manually) for each component assembly.

However, ideally, component connectors are generic, self-contained and preexist. They can be, like components, deployed into an assembly to make component connections. For these component models, the Deployment Contracts Analyser offers support in defining assemblies using pre-existing composition operators.

Currently, only component model with Exogenous Connectors [13, 12] employs special composition operators for composing components that possess these properties. In future, other component models supporting deployment time composition, and having generic and self-contained component composition operators may arise.

Since currently only component model with Exogenous Connectors has pre-existing composition operators, we can demonstrate the ability of the Deployment Contracts Analyser to support assemblies of pre-existing components and composition operators using this component model. On the right hand side in Figure 17 loaded exogenous connectors are shown.

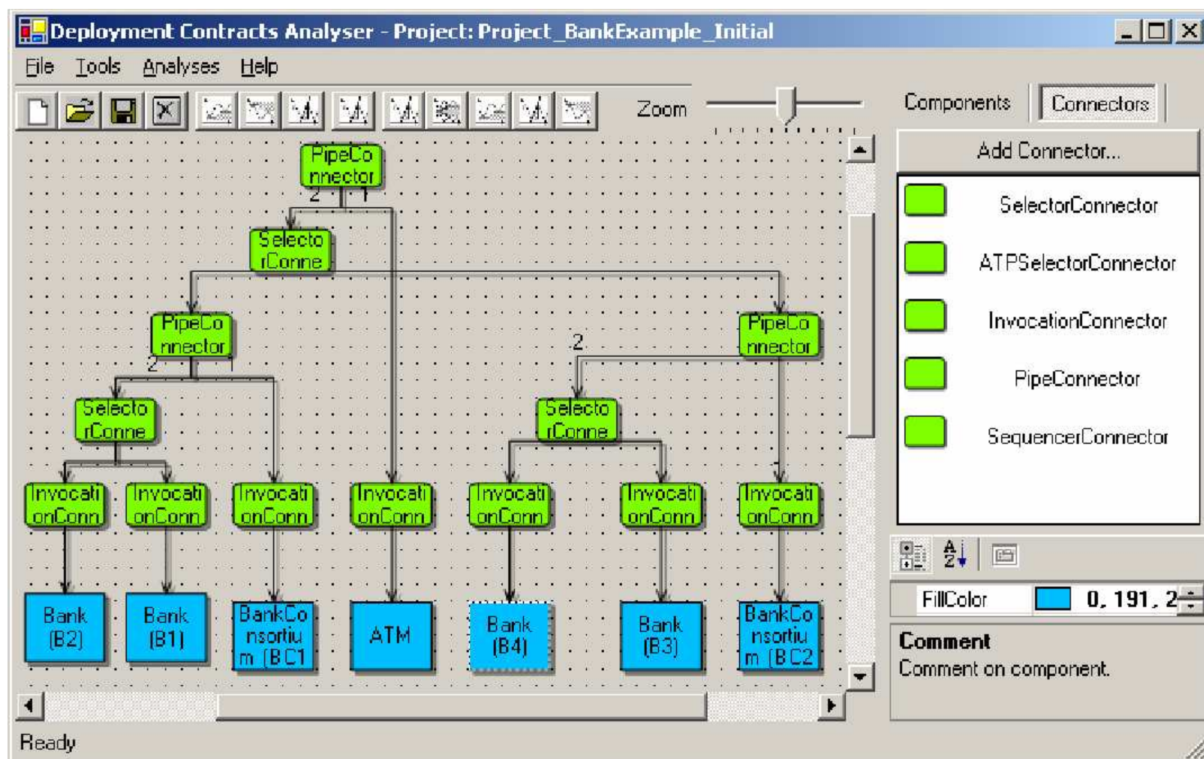


Figure 17: Loading Component Connectors.

They are loaded in the same way as components. Also like components, they can be dragged from the tab card on the right hand side in Figure 17 and dragged on to the surface on the left hand side. Having components and connectors on the surface, an architecture of the assembly can be visually created. All the other steps in the workflow of the Deployment Contracts Analyser from Section 4 remain the same.

4.1.1 Support for Automated Component Composition

In current component models supporting deployment time composition, component composition is not automated. Ideally, however, component composition is automated.

Component Model with Exogenous Connector has a generic container [7, 9] for automatic component composition at runtime. In order for the generic container to automatically compose components at runtime, it requires a composition plan, which has to be created at deployment time. The Deployment Contracts Analyser supports automatic generation of the composition plan based on the assembly architecture of components and exogenous connectors.

Figure 18 shows an assembly of components and exogenous connectors and illustrates how the system developer can have the composition plan for the assembly generated.

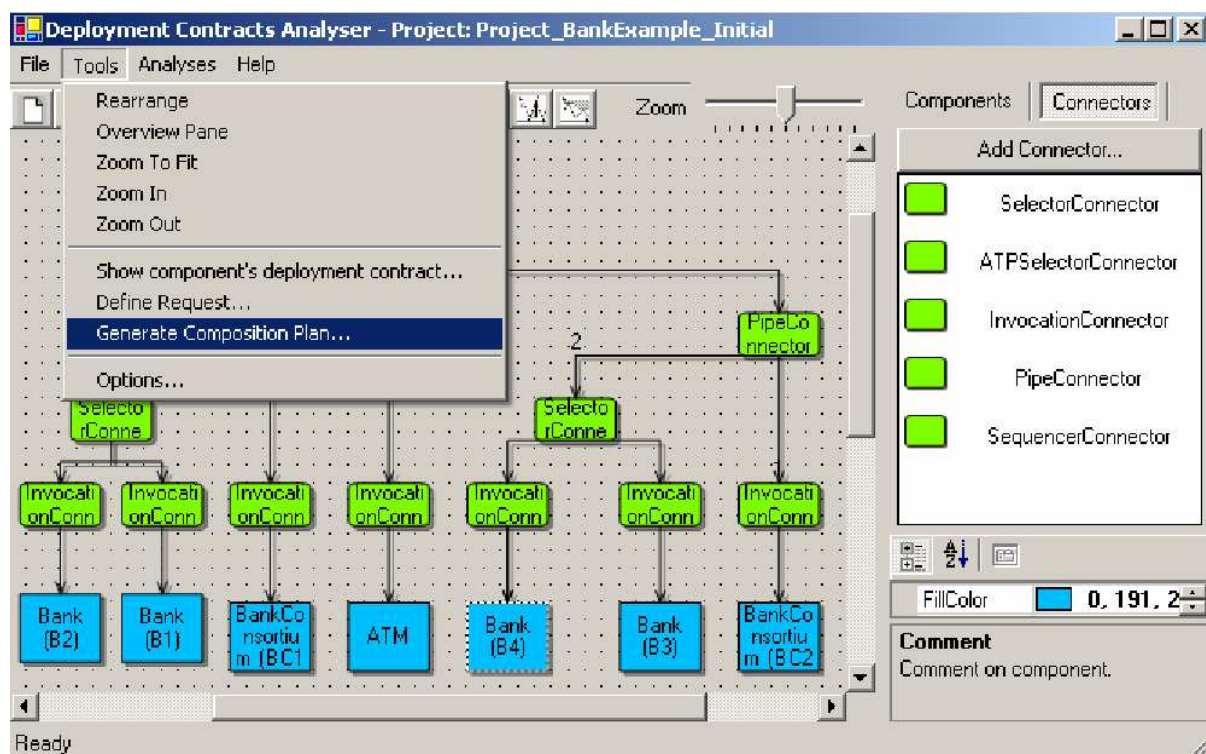


Figure 18: Generating Composition Plan.

After the composition plan has been generated by the Deployment Contracts Analyser, the generic container for components and exogenous connectors can construct the assembly following the composition plan, and execute it.

Thus, the Deployment Contracts Analyser has extensive support for the component model with exogenous connectors: component composition can be checked, and if no problems have been found, a composition plan for the assembly can be generated, following which the generic container can automatically construct and execute the assembly.

5 Examples of Deployment Contracts Analysis

In this section we show how the Deployment Contracts Analyser from Section 4 can be used to automatically spot conflicts with component assemblies from Section 2.

5.1 Spotting conflicts due to absence of resources in the execution environment required by components in an assembly

In this section we show how the conflicts from Section 2.1 can be spotted using deployment contracts of components and their analysis.

5.1.1 Example 1

Consider Example 1 shown in Figure 19. The example was schematically shown in Section 2.1.

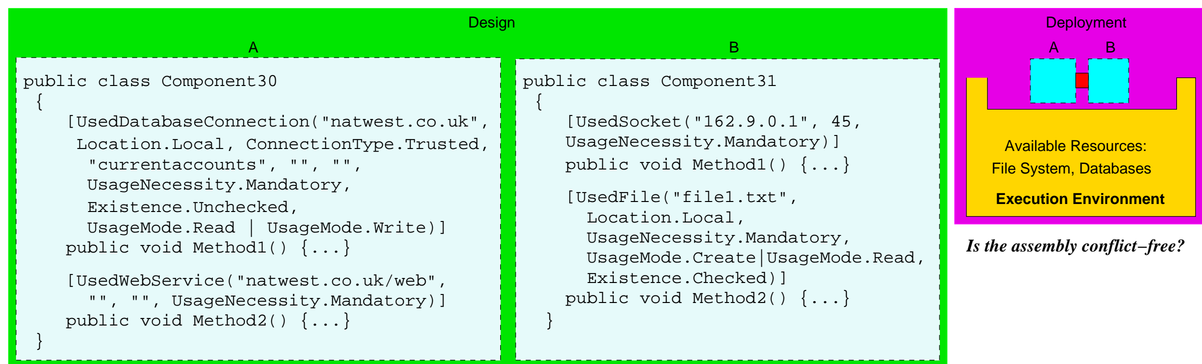


Figure 19: Example 1.

Component A is designed in a way that it makes use of a database connection in method “A.Method1” and of a web service in method “A.Method2”. This can be seen from its deployment contract in Figure 19.

Component B is designed in a way that it makes use of a socket in method “B.Method1” and of a file in method “B.Method2”. This can be seen from its deployment contract in Figure 19.

Suppose at deployment time, an assembly AB is created. In the assembly, components’ methods are connected so that there are two connections:

- Connection 1: method “A.Method1” is called prior to the method “B.Method1”,
- Connection 2: method “A.Method2” is called prior to the method “B.Method2”

In the execution environment of the assembly file system and databases are available. The type of environment, desktop or web, is irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 20.

```
---------
Check Each Component's Deployment Contract against Execution Environment's Resources:

Component 'Component30':
Method-level attribute 'UsedDatabaseConnection' on method 'Method1': [UsedDatabaseConnection(DatabaseServerName=natwest.co.uk, Location=Local,
ConnectionType=Trusted, DatabaseName=currentaccounts, UserName=, Password=, UsageNecessity=Mandatory, Existence=Unchecked,
UsageMode=Read, Write)]
Local Databases required. Local Databases are available in the assembly's target execution environment. OK
Method-level attribute 'UsedWebService' on method 'Method2': [UsedWebService(Url=natwest.co.uk/web, UserName=, Pwd=, UsageNecessity=Mandatory)]
Network required. Network is not available in the assembly's target execution environment. ERROR
Summary: Component 'Component30' - ERRORS: 1, WARNINGS: 0, HINTS: 0

Component 'Component31':
Method-level attribute 'UsedSocket' on method 'Method1': [UsedSocket(IPAdress=162.9.0.1, PortNumber=45, UsageNecessity=Mandatory)]
Network required. Network is not available in the assembly's target execution environment. ERROR
Method-level attribute 'UsedFile' on method 'Method2': [UsedFile(FullName=file1.txt, Location=Local, UsageNecessity=Mandatory, UsageMode=Read, Create,
Existence=Checked)]
File System required. File System is available in the assembly's target execution environment. OK
Summary: Component 'Component31' - ERRORS: 1, WARNINGS: 0, HINTS: 0
---------
```

Figure 20: Deployment contracts analysis for the Example 1.

For the component A (Component30 in Figure 20), the DCA finds out that the web service cannot be accessed in the assembly's execution environment due to absence of network.

For the component B (Component30 in Figure 20), the DCA finds out that the socket is not effectively usable in the assembly's execution environment due to absence of network.

Thus, deployment contracts analysis of the assembly AB has shown 2 errors. Therefore, the assembly AB is not conflict-free and cannot execute safely at runtime.

5.2 Spotting conflicts due to contentious use of available resources by components in an assembly

In this section we show how the conflicts from Section 2.2 can be spotted using deployment contracts of components and their analysis.

5.2.1 Example 2

Consider Example 2 shown in Figure 21. The example was schematically shown in Section 2.2.

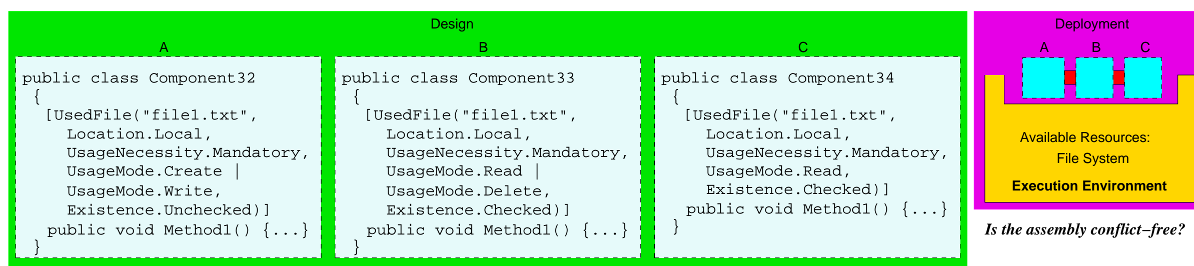


Figure 21: Example 2.

Component A is designed in a way that it makes use of the file "file1.txt" in method "A.Method1". The file must be local to the component, i.e. it must reside in the same directory where the component itself is. The usage of the file is mandatory for the component. The component creates the file and writes into it. Moreover, the component does not check whether

the file exists or not before creating it. This can be seen from its deployment contract in Figure 21.

Component B is designed in a way that it also makes use of the file “file1.txt” in method “B.Method1”. The file must be local to the component, i.e. it must reside in the same directory where the component itself is. The usage of the file is mandatory for the component. The component reads from the file and deletes it. Moreover, the component checks whether the file exists before reading from it. This can be seen from its deployment contract in Figure 21.

Component C is designed in a way that it also makes use of the file “file1.txt” in method “C.Method1”. The file must be local to the component, i.e. it must reside in the same directory where the component itself is. The usage of the file is mandatory for the component. The component reads from the file. Moreover, the component checks whether the file exists before reading from it. This can be seen from its deployment contract in Figure 21.

Suppose at deployment time, an assembly ABC is created. In the assembly, components' methods are connected so that there one connection:

- Connection 1: method “A.Method1” is called prior to the method “B.Method1” which is called prior to the method “C.Method1”

In the execution environment of the assembly file system is available. The type of environment, desktop or web, is irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly ABC is shown in Figure 22.

```
----------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Usage of Resources in the Execution Environment:

Component Connection 'Request 1':

Component 'Component32', Attribute 'UsedFile': [UsedFile(FullName=file1.txt, Location=Local, UsageNecessity=Mandatory, UsageMode=Write, Create,
Existence=Unchecked)]
It is unknown whether file exists or not. file is created afresh, which may cause data loss. - WARNING If file exists but cannot be created or used the
component will fail to execute since its usage is mandatory. - HINT. If file does not exist and cannot be created or used the component will fail to execute
since its usage is mandatory. - HINT
In presence of multiple threads operating concurrently in this and the next component, Readers and Writers problem may occur. Please, refer to a concurrent
programming language to check it, if required. - HINT

Component 'Component33', Attribute 'UsedFile': [UsedFile(FullName=file1.txt, Location=Local, UsageNecessity=Mandatory, UsageMode=Read, Delete,
Existence=Checked)]
file exists.  If file cannot be used or deleted the component will fail to execute since its usage is mandatory. - HINT

Component 'Component34', Attribute 'UsedFile': [UsedFile(FullName=file1.txt, Location=Local, UsageNecessity=Mandatory, UsageMode=Read,
Existence=Checked)]
file does not exist. file cannot be used. The component will fail to execute since its usage is mandatory. - ERROR
Summary: Component Connection 'Request 1' - ERRORS: 1, WARNINGS: 1, HINTS: 4
----------
```

Figure 22: Deployment contracts analysis for the Example 2.

For the component A (Component32 in Figure 22), the DCA finds out that the component does not check the file “file1.txt” for existence. Therefore, if the file exists before the component A creates it, it is created afresh, which may cause data loss. Moreover, the DCA finds out that the usage of the file is mandatory for the component A. Therefore, if the file cannot be created or used, the component will fail to execute. Finally, the DCA finds out that the component A writes into the file and the component B reads from it. Therefore, in the presence of multiple threads operating concurrently in these components, Readers and Writers problem [21] may occur.

For the component B (Component33 in Figure 22), the DCA finds out that the usage of the file “file1.txt” is mandatory. Therefore, if the file cannot be read from or deleted, the component

will fail to execute.

For the component C (Component34 in Figure 22), the DCA finds out that the file "file1.txt" is mandatory. The component reads from the file. However, it is deleted by the previous component B. Therefore, the component C will fail to execute.

Thus, deployment contracts analysis of the assembly ABC has shown 1 error and 1 warning. The error is fatal. The component B has to be replaced by another one in the assembly ABC. The assembly ABC is not conflict-free and cannot execute safely at runtime.

5.2.2 Example 3

Consider Example 3 shown in Figure 23. The example was schematically shown in Section 2.2.

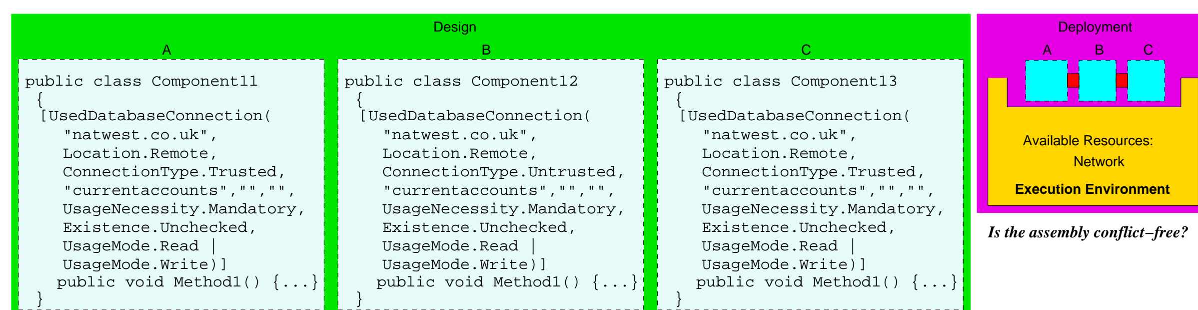


Figure 23: Example 3.

Component A is designed in a way that it makes use of a database connection to the database "natwest.co.uk". The database is installed remotely on a separate server. The component uses an encrypted database connection to the database. The database connection is mandatory for the component. The component does not check whether the connection exists or not. It assumes that the connection is there, ready to be used. The component uses the connection in Read/Write mode. This can be seen from its deployment contract in Figure 23.

Component B is designed in a way that it makes use of a database connection to the database "natwest.co.uk". The database is installed remotely on a separate server. The component uses an unencrypted database connection to the database. The database connection is mandatory for the component. The component does not check whether the connection exists or not. It assumes that the connection is there, ready to be used. The component uses the connection in Read/Write mode. This can be seen from its deployment contract in Figure 23.

Component C is designed in a way that it makes use of a database connection to the database "natwest.co.uk". The database is installed remotely on a separate server. The component uses an encrypted database connection to the database. The database connection is mandatory for the component. The component does not check whether the connection exists or not. It assumes that the connection is there, ready to be used. The component uses the connection in Read/Write mode. This can be seen from its deployment contract in Figure 23.

Suppose at deployment time, an assembly ABC is created. In the assembly, components' methods are connected so that there one connection:

- Connection 1: method "A.Method1" is called prior to the method "B.Method1" which is called prior to the method "C.Method1"

In the execution environment of the assembly network is available. The type of environment, desktop or web, is irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly ABC is shown in Figure 24.

```
----------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Usage of Resources in the Execution Environment:

Component Connection 'Request 1':

Component 'Component11', Attribute 'UsedDatabaseConnection': [UsedDatabaseConnection(DatabaseServerName=natwest.co.uk, Location=Remote,
ConnectionType=Trusted, DatabaseName=currentaccounts, UserName=, Password=, UsageNecessity=Mandatory, Existence=Unchecked,
UsageMode=Read, Write)]
It is unknown whether database connection exists or not. If database connection exists but cannot be used the component will fail to execute since its usage
is mandatory. - HINT If database connection does not exist and cannot be used the component will fail to execute since its usage is mandatory. - WARNING
In presence of multiple threads operating concurrently in this and the next component, Readers and Writers problem may occur. Please, refer to a concurrent
programming language to check it, if required. - HINT

Component 'Component12', Attribute 'UsedDatabaseConnection': [UsedDatabaseConnection(DatabaseServerName=natwest.co.uk, Location=Remote,
ConnectionType=Untrusted, DatabaseName=currentaccounts, UserName=, Password=, UsageNecessity=Mandatory, Existence=Unchecked,
UsageMode=Read, Write)]
It is unknown whether database connection exists or not. If database connection exists but cannot be used the component will fail to execute since its usage
is mandatory. - HINT If database connection does not exist and cannot be used the component will fail to execute since its usage is mandatory. - WARNING
In presence of multiple threads operating concurrently in this and the next component, Readers and Writers problem may occur. Please, refer to a concurrent
programming language to check it, if required. - HINT

Component 'Component13', Attribute 'UsedDatabaseConnection': [UsedDatabaseConnection(DatabaseServerName=natwest.co.uk, Location=Remote,
ConnectionType=Trusted, DatabaseName=currentaccounts, UserName=, Password=, UsageNecessity=Mandatory, Existence=Unchecked,
UsageMode=Read, Write)]
It is unknown whether database connection exists or not. If database connection exists but cannot be used the component will fail to execute since its usage
is mandatory. - HINT If database connection does not exist and cannot be used the component will fail to execute since its usage is mandatory. - WARNING

Not all components in the connection use a trusted database connection. There may be a security problem with the assembly. - WARNING
Summary: Component Connection 'Request 1' - ERRORS: 0, WARNINGS: 4, HINTS: 5
----------
```

Figure 24: Deployment contracts analysis for the Example 3.

For the component A (Component11 in Figure 24), the DCA finds out that the component does not check the database connection for existence. Therefore, if the connection cannot be used, the component will fail to execute since it is mandatory for the component.

For the components B and C (Component12 in Figure 24), the DCA finds out the same as for the component A.

Additionally, the DCA finds out that not all components in the connection use encrypted database connection. Assume that the system has tight security requirements. For that system, the assembly ABC is insecure. The component B has to be replaced by another one using a trusted database connection.

5.2.3 Example 4

Consider Example 4 shown in Figure 25. The example was schematically shown in Section 2.2.

Component A is designed in a way that its method "A.Method1" creates and writes the environment variable "SEARCH_PATH". The component does not check whether the environment variable exists before using it. This can be seen from the component A's deployment contract in Figure 25.

Component B is designed in a way that its method "B.Method1" also sets and reads the same environment variable "SEARCH_PATH". It also does not check whether the environment variable exists before using it. This can be seen from the component B's deployment contract in Figure 25.

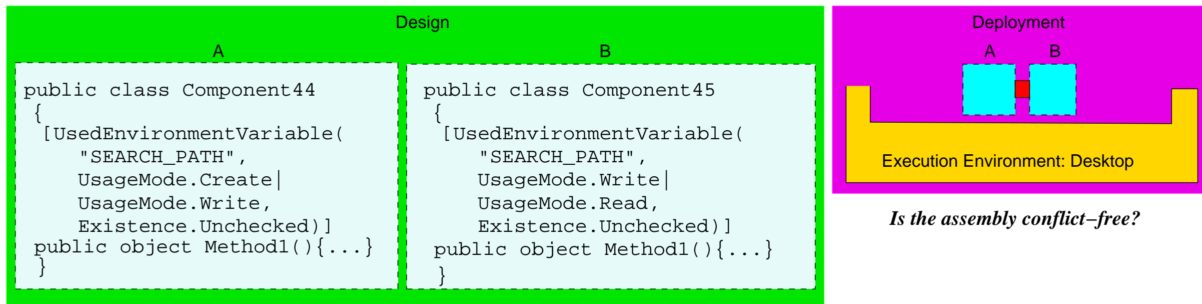


Figure 25: Example 4.

Suppose at deployment time, an assembly AB is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method "A.Method1" invokes method "B.Method1".

The type of the assembly's execution environment is desktop. Resources available in the environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 26.

```
---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Usage of Resources in the Execution Environment:

Component Connection 'Request 1':
In presence of multiple threads operating concurrently in this and the next component, Readers and Writers problem may occur. Please, refer to a concurrent
programming language to check it, if required. - HINT

Components in the assembly set the same environment variable. - WARNING
Summary: Component Connection 'Request 1' - ERRORS: 0, WARNINGS: 1, HINTS: 1
---------
```

Figure 26: Deployment contracts analysis for the Example 4.

For the connection specified above, the DCA finds out that components A and component B set the same environment variable. This is suspicious since most probably the component will interfere with each other by setting the environment variable to the same value and reading each other's, i.e. wrong, values of the environment variables.

It is better for the system developer to find components that do not set the same environment variables. Alternatively, the assembly can be built and run to see whether the fact that components A and B set the same environment variable hinders their execution.

5.3 Spotting conflicts due to incompatible threading models of components in an assembly

In this section we show how the conflicts from Section 2.3 can be spotted using deployment contracts of components and their analysis.

5.3.1 Example 5

Consider Example 5 shown in Figure 27. The example was schematically shown in Section 2.3.

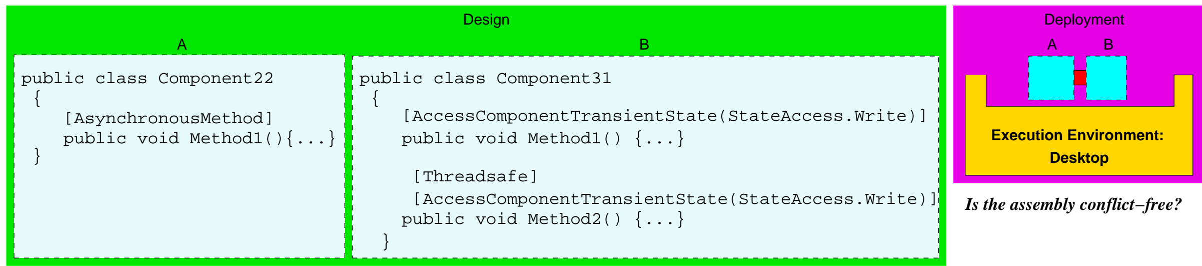


Figure 27: Example 5.

Component A is designed in a way that its method "A.Method1" is asynchronous. This can be seen from its deployment contract in Figure 27.

Component B is designed in a way that its method "B.Method1" accesses component's transient state in write mode. The method "B.Method2" also accesses component's transient state in write mode. In addition, the method "B.Method2" is thread-safe. This can be seen from its deployment contract in Figure 27.

Suppose at deployment time, an assembly AB is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method "A.Method1" invokes method "B.Method1". The connection is recurrent.

The type of the assembly's execution environment is desktop. Resources available in the environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 28.

```
---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the Execution Environment:

Component Connection 'Request 1':

Component 'Component23' accesses its state in not read-only mode. The state can be concurrently accessed by multiple threads from within the component 'Component22' but is unprotected from concurrent access by multiple threads. - ERROR
Summary: Component Connection 'Request 1' - ERRORS: 1, WARNINGS: 0, HINTS: 0
---------
```

Figure 28: Deployment contracts analysis for the Example 5.

For the connection specified above, the DCA finds out that component A (Component22 in Figure 28) accesses component B (Component23 in Figure 28) from an internally created thread since the method "A.Method1" is asynchronous [6]. Furthermore, the connection is recurrent. Therefore, the method "B.Method1" can be concurrently accessed by multiple threads from within the method "A.Method1". Since, the method "B.Method1" is not thread-safe and manipulates component's state, state corruption problem will occur in the component B. The DCA issues an error in this case.

Note that the method "B.Method2" also accesses B's state. In contrast to the method "B.Method1", it is thread-safe. If it was connected to the method "A.Method1", no state corruption problem would occur.

Finally, deployment contracts analysis of the assembly AB has shown 1 errors. Therefore, the assembly AB is not conflict-free and cannot execute safely at runtime.

5.3.2 Example 6

Consider Example 6 shown in Figure 29. The example was schematically shown in Section 2.3.

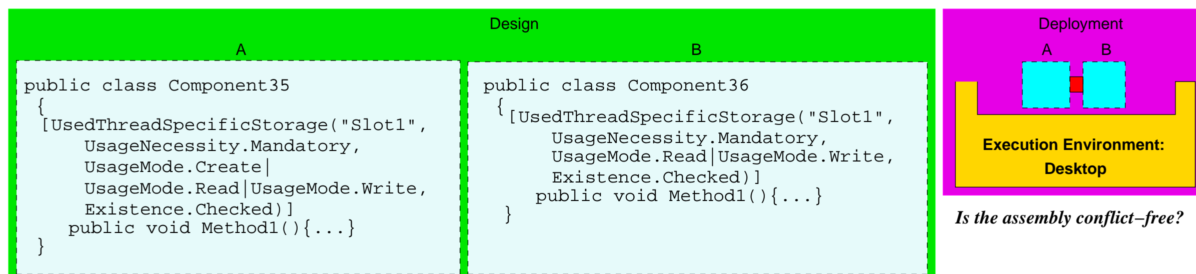


Figure 29: Example 6.

Component A is designed in a way that its method "A.Method1" makes use of thread-specific storage [18]. In particular, it uses the slot "Slot1". The usage of the slot is mandatory for the component. It creates the slot as well as reads from and writes to it. This can be seen from component's deployment contract in Figure 29.

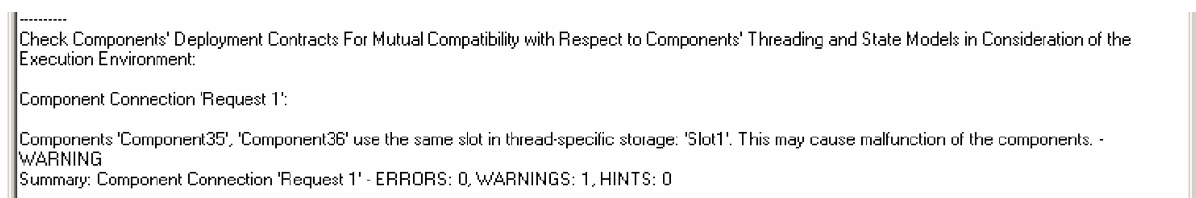
Component B is designed in a way that it also makes use of the slot "Slot1" in thread-specific storage. The usage of the slot is mandatory for the component. It reads from the slot and writes to it. This can be seen from component's deployment contract in Figure 29.

Suppose at deployment time, an assembly AB is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method "A.Method1" is invoked prior to method "B.Method1".

The type of the assembly's execution environment is desktop. Resources available in the environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 30.



Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the Execution Environment:

Component Connection 'Request 1':

Components 'Component35', 'Component36' use the same slot in thread-specific storage: 'Slot1'. This may cause malfunction of the components. - WARNING
Summary: Component Connection 'Request 1' - ERRORS: 0, WARNINGS: 1, HINTS: 0

Figure 30: Deployment contracts analysis for the Example 6.

For the connection specified above, the DCA finds out that component A (Component35 in Figure 30) and component B (Component36 in Figure 30) both use the same slot in the thread-specific storage – "Slot1". Both components read from and write to the slot. This is a suspicious situation where the components may write completely different pieces of information to the same slot of thread-specific storage, and thus may interfere with each other. However, to be completely sure whether the assembly will fail to execute at runtime, it has to be built and run. The DCA cannot completely investigate this potential issue.

5.3.3 Example 7

Consider Example 7 shown in Figure 31.

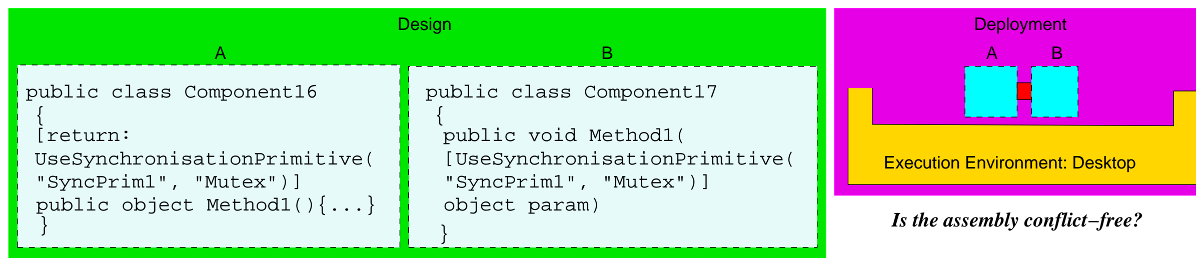


Figure 31: Example 7.

Component A is designed in a way that its method "A.Method1" a handle to a synchronisation primitive, namely to the Mutex "SyncPrim1". This can be seen from component's deployment contract in Figure 31.

Component B is designed in a way that its method "B.Method1" accepts a handle to the Mutex "SyncPrim1" as a parameter. This can be seen from component's deployment contract in Figure 31.

Suppose at deployment time, an assembly AB is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method "A.Method1" is invoked prior to method "B.Method1".

The type of the assembly's execution environment is desktop. Resources available in the environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 32.

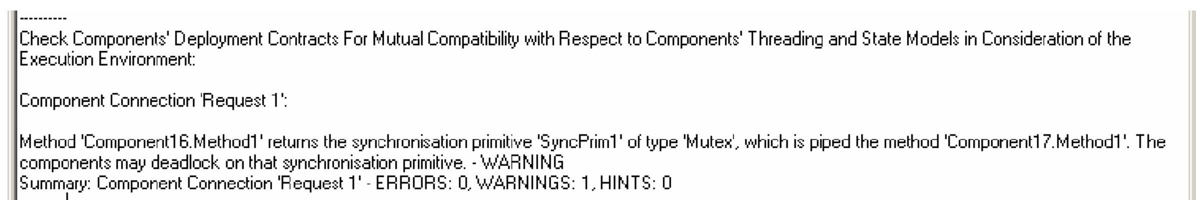


Figure 32: Deployment contracts analysis for the Example 7.

For the connection specified above, the DCA finds out that components A (Component16 in Figure 32) and B (Component17 in Figure 30) exchange the handle to the mutex "SyncPrim1". Having the same synchronisation primitive in different components is one of the prerequisites for a deadlock. Therefore, the DCA can flag this potential issue with the components to the system developer. However, to be completely sure whether the assembly will fail to execute at runtime, it has to be built and run. This cannot be completely investigated by the DCA.

5.4 Spotting conflicts due to incompatible threading model of a component and concurrency management of the execution environment

In this section we show how the conflicts from Section 2.4 can be spotted using deployment contracts of components and their analysis.

5.4.1 Example 8

Consider Example 8 shown in Figure 33. The example was schematically shown in Section 2.4.

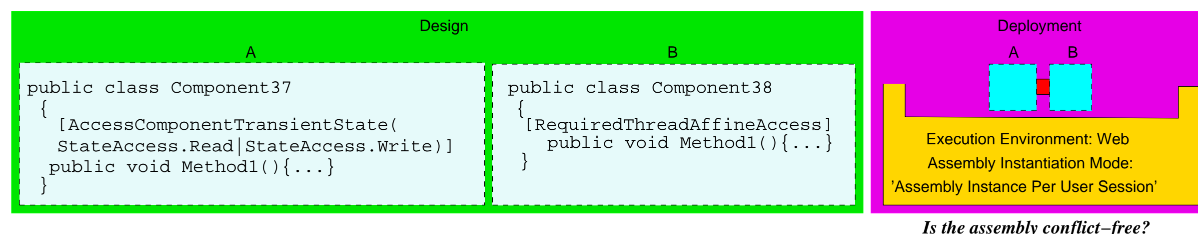


Figure 33: Example 8.

Component A is designed in a way that its method "A.Method1" accesses component's transient state in Read/Write mode. This can be seen from its deployment contract in Figure 33.

Component B is designed in a way that its method "B.Method1" requires thread affine access. This can be seen from its deployment contract in Figure 33.

Suppose at deployment time, an assembly AB is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method "A.Method1" is invoked prior to the method "B.Method1",

The assembly AB is deployed into the web environment with assembly instantiation mode 'assembly instance per user session'. Resource available in the execution environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 34.

```
---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the
Execution Environment:

Component Connection 'Request 1':

Component 'Component38' requires thread affine access. It is deployed to the web environment with assembly instantiation mode 'assembly instance per user
session'. Multiple threads induced by the web environment will access the component. Therefore, thread affinity cannot be guaranteed. - ERROR
The assembly is deployed into the web environment with assembly instantiation mode 'assembly instance per user session'. State is only retained for a user
session. Check if it is appropriate. - WARNING
Summary: Component Connection 'Request 1' - ERRORS: 1, WARNINGS: 1, HINTS: 0
---------
```

Figure 34: Deployment contracts analysis for the Example 8.

For the component B (Component38 in Figure 34), the DCA finds out that the requirement of the method "A.Method1" cannot be satisfied due to the concurrency management of the environment the assembly AB is deployed to, namely absence of thread affinity of the main thread.

Moreover, in this environment, state is only retained for the duration of a user session. This is relevant for the component A (Component37 in Figure 34). Assume that this is acceptable for the system the system developer is building.

Deployment contracts analysis of the assembly AB has shown 1 error. Therefore, the assembly AB is not conflict-free and cannot execute safely at runtime. Component B has to be replaced by another one in the assembly AB.

5.4.2 Example 9

Consider the assembly from the Section 5.4.1 deployed into the web environment with a different assembly instantiation mode (Figure 35): (The example was schematically shown in Section 2.4.)

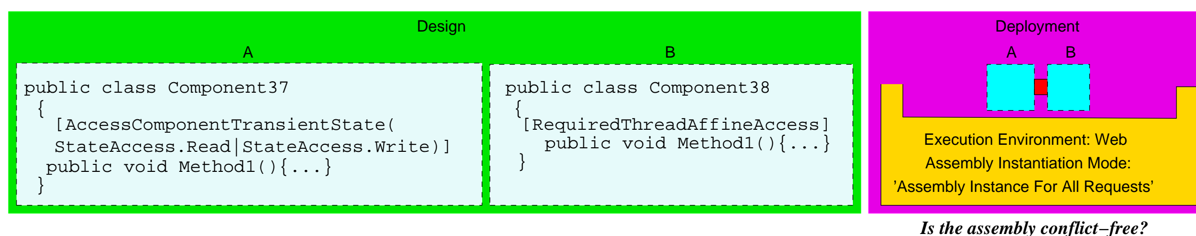


Figure 35: Example 9.

The assembly AB is deployed into the web environment with assembly instantiation mode 'assembly instance for all requests'. Resource available in the execution environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 36.

```
---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the
Execution Environment:

Component Connection 'Request 1':

Component 'Component37' accesses its state in not read-only mode. The state can be concurrently accessed by multiple threads by the web environment
but is unprotected from concurrent access by multiple threads. - ERROR

Component 'Component38' requires thread affine access to one of its parts but can be accessed concurrently by multiple threads imposed by the web
environment. - ERROR
Summary: Component Connection 'Request 1' - ERRORS: 2, WARNINGS: 0, HINTS: 0
---------
```

Figure 36: Deployment contracts analysis for the Example 9.

For the component A (Component37 in Figure 36), the DCA finds out that component's state will be accessed concurrently in the assembly's execution environment. Since the state is unprotected from concurrent access by multiple threads, state corruption problem will occur.

For the component B (Component38 in Figure 36), the DCA finds out that the requirement of the method "A.Method1" cannot be satisfied due to the concurrency management of the environment the assembly AB is deployed to, namely concurrent access of the assembly by multiple threads.

Deployment contracts analysis of the assembly AB has shown 2 errors. Therefore, the assembly AB is not conflict-free and cannot execute safely at runtime.

5.4.3 Example 10

Consider the assembly from the Section 5.4.1 deployed into the desktop environment(Figure 37):

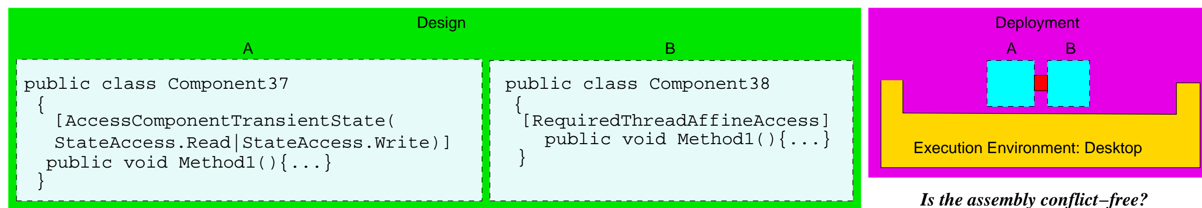


Figure 37: Example 10.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 38.

```
---------
Check Each Component's Deployment Contract against Execution Environment's Resources:

Component 'Component37':
Summary: Component 'Component37' - OK

Component 'Component38':
Summary: Component 'Component38' - OK
---------

---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Usage of Resources in the Execution Environment:

Component Connection 'Request 1':
Summary: Component Connection 'Request 1' - OK
---------

---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the Execution Environment:

Component Connection 'Request 1':
Summary: Component Connection 'Request 1' - OK
---------
```

Figure 38: Deployment contracts analysis for the Example 10.

Neither for the component A (Component37 in Figure 38), nor for the component B (Component38 in Figure 38) has the DCA found any problem.

Deployment contracts analysis of the assembly AB has not shown any errors or warnings. Therefore, the assembly AB is conflict-free and can execute safely at runtime.

5.5 Spotting conflicts due to incompatible state model of a component and state management of the execution environment

In this section we show how the conflicts from Section 2.5 can be spotted using deployment contracts of components and their analysis.

5.5.1 Example 11

Consider Example 11 shown in Figure 39. The example was schematically shown in Section 2.5.

Component A is designed in a way that it accesses component's state in method "A.Method1" in Read/Write mode. This can be seen from its deployment contract in Figure 39.

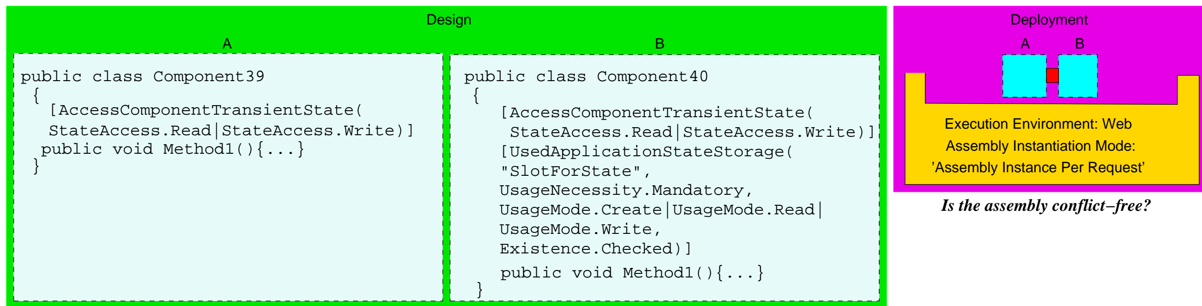


Figure 39: Example 11.

Component B is designed in a way that it accesses component's state in method "B.Method1" in Read/Write mode. Furthermore, the state is retain in the application state storage. This can be seen from its deployment contract in Figure 39.

Suppose at deployment time, an assembly AB is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method "A.Method1" is called prior to the method "B.Method1",

The type of environment is web. Assembly instantiation mode is 'assembly instance per request'. Resource available in the environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 40.

```
---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the
Execution Environment:

Component Connection 'Request 1':

Component 'Component39' is stateful. It is deployed into the web environment with assembly instantiation mode 'assembly instance per request', where state
retention issues exist. However, the component makes use neither of session nor of application state storage. Therefore, it will lose its state on each request
to the assembly. - ERROR
Summary: Component Connection 'Request 1' - ERRORS: 1, WARNINGS: 0, HINTS: 0
---------
```

Figure 40: Deployment contracts analysis for the Example 11.

For the component A (Component39 in Figure 40), the DCA finds out that in the assembly's execution environment no state retention is done. Since the component A does not retain its state in a state storage, it will lose it state after each request and therefore fail to execute properly.

Note that the component B stores its state in an application state storage. Therefore, the DCA does find any issue with it.

Thus, deployment contracts analysis of the assembly AB has shown 1 error. Therefore, the assembly AB is not conflict-free and cannot execute safely at runtime. The component A has to be replaced by another one in the assembly.

Note that if the assembly AB would be deployed into the web environment with assembly instantiation mode 'pool of synchronised assembly instances for all requests', the results of the DCA analysis would be the same.

5.6 Spotting combined conflicts

In this section we show how the conflicts from Sections 5.1 – 5.5 can occur in combination in component assemblies, and how they can be spotted using deployment contracts of components and their analysis.

5.6.1 Example 12

Consider Example 12 shown in Figure 41.

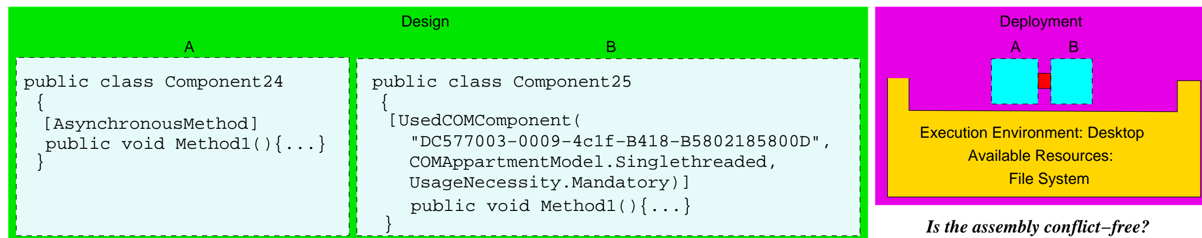


Figure 41: Example 12.

Component A is designed in a way that its method "A.Method1" is asynchronous. This can be seen from its deployment contract in Figure 41.

Component B is designed in a way that its method "B.Method1" makes use of a COM component. The COM component requires a single threaded apartment. The usage of the COM component is mandatory for the component B. This can be seen from its deployment contract in Figure 41.

Suppose at deployment time, an assembly AB is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method "A.Method1" invokes method "B.Method1". The connection is recurrent.

The assembly is deployed into the desktop environment. In the environment, file system is available.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 42.

```
----------
Check Each Component's Deployment Contract against Execution Environment's Resources:

Component 'Component24':
Summary: Component 'Component24' - OK

Component 'Component25':
Component-level attribute 'UsedCOMComponent': [UsedCOMComponent(Guid=DC577003-0009-4c1f-B418-B5802185800D,
COMAppartmentModel=Singlethreaded, UsageNecessity=Mandatory)]
File System required. File System is available in the assembly's target execution environment. OK
Component-level attribute 'UsedCOMComponent': [UsedCOMComponent(Guid=DC577003-0009-4c1f-B418-B5802185800D,
COMAppartmentModel=Singlethreaded, UsageNecessity=Mandatory)]
Local Registry Storage required. Local Registry Storage is not available in the assembly's target execution environment. ERROR
Summary: Component 'Component25' - ERRORS: 1, WARNINGS: 0, HINTS: 0
----------

----------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Usage of Resources in the Execution Environment:

Component Connection 'Request 1':
Summary: Component Connection 'Request 1' - OK
----------

----------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the Execution
Environment:

Component Connection 'Request 1':

Component 'Component25' requires thread affine access to one of its parts but can be accessed concurrently by component 'Component24'. - ERROR
Summary: Component Connection 'Request 1' - ERRORS: 1, WARNINGS: 0, HINTS: 0
----------
```

Figure 42: Deployment contracts analysis for the Example 12.

With the component A (Component24 in Figure 42), the DCA does not find any problem.

For the component B (Component25 in Figure 42), the DCA finds out that the component requires access to local (Windows) registry because it makes use of a COM component, which must be registered there. Since the registry is not available in assembly's execution environment, the component B will fail to execute in the environment.

Moreover, component A invokes component B concurrently. Since component B requires thread affine access due to use of a COM component requiring single threaded apartment, it will fail to execute assembled with component A.

Thus, deployment contracts analysis of the assembly AB has shown 2 errors. Therefore, the assembly AB is not conflict-free and cannot execute safely at runtime. The component A has to be replaced by another one in the assembly. Moreover, assembly's execution environment must offer access to local (Windows) registry.

5.6.2 Example 13

Consider example from Section 5.6.1 again. This time the assembly is put into the web environment (Figure 43).

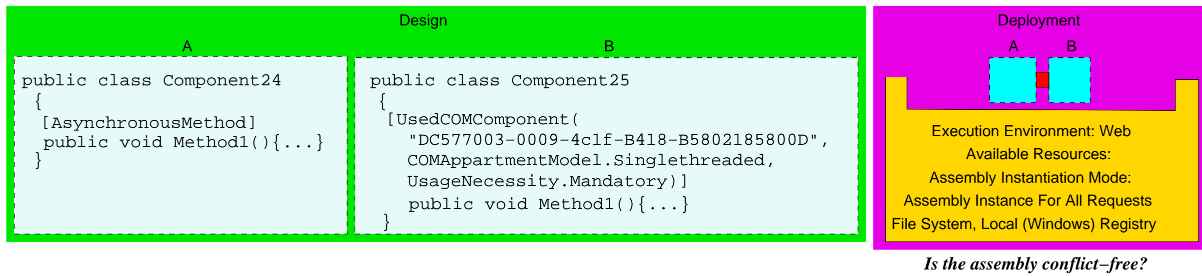


Figure 43: Example 13.

The type of the assembly's execution environment is web. Assembly instantiation mode is 'assembly instance for all requests'. Resources available in the environment are file system and local (Windows) registry.

Deployment contracts analysis performed by the DCA for the assembly AB is shown in Figure 44.

Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the Execution Environment:

Component Connection 'Request 1':

Component 'Component25' requires thread affine access to one of its parts but can be accessed concurrently by component 'Component24'. - ERROR

Component 'Component25' requires thread affine access. It is deployed to the web environment with assembly instantiation mode 'assembly instance for all requests'. Multiple threads induced by the web environment will access the component. Therefore, thread affinity cannot be guaranteed. - ERROR
Summary: Component Connection 'Request 1' - ERRORS: 2, WARNINGS: 0, HINTS: 0

Figure 44: Deployment contracts analysis for the Example 13.

The problem with the assembly coming from the fact the execution environment does not offer access to local registry storage appeared in Section 5.6.1 disappeared here.

However, another problem has arisen. In this execution environment, component B will be accessed concurrently by multiple threads induced by the environment. This will lead to failure of component B since it required thread affine access.

Therefore, the assembly AB is not conflict-free and cannot execute safely at runtime.

5.6.3 Example 14

Consider Example 14 shown in Figure 45.

Component A is designed in a way that its method "A.Method1" returns a handle, which requires thread affinity. This can be seen from its deployment contract in Figure 45.

Component B is designed in a way that its method "B.Method1" is asynchronous. This can be seen from its deployment contract in Figure 45.

Component C is designed in a way that all its methods make use of a message queue. The usage of the message queue is mandatory for the component. It creates the queue as well as reads from and writes to it. Moreover, the component checks the queue for existence. Component C's method "C.Method1" make use of an XML file, whose location is remote. The usage of the XML file is optional for the component. Its existence is checked. This can be seen from component's deployment contract in Figure 45.

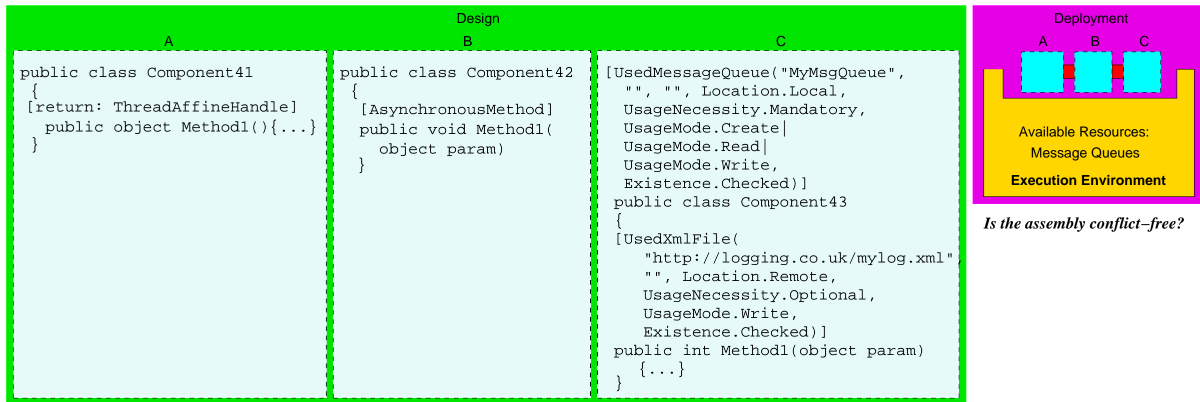


Figure 45: Example 14.

Suppose at deployment time, an assembly ABC is created. In the assembly, components' methods are connected so that there is one connection:

- Connection 1: method's "A.Method1" return value is piped as parameter into the method "B.Method1", which invokes the method "C.Method1". The connection is recurrent.

The assembly is deployed into the desktop environment. In the environment, message queues are available.

Deployment contracts analysis performed by the DCA for the assembly ABC is shown in Figure 46.

```
Component 'Component43':
Component-level attribute 'UsedMessageQueue': [UsedMessageQueue(Name=MyMsgQueue, UserName=, UserPwd=, Location=Local,
UsageNecessity=Mandatory, UsageMode=Read, Write, Create, Existence=Checked)]
Local Message Queues required. Local Message Queues are available in the assembly's target execution environment. OK
Method-level attribute 'UsedXmlFile' on method 'Method1': [UsedXmlFile(FullFileName=http://logging.co.uk/mylog.xml, SchemaFullFileName=, Location=Remote,
UsageNecessity=Optional, UsageMode=Write, Existence=Checked)]
Network required. Network are not available in the specified target execution environment. However, the use of Network is optional for the component.
Therefore, the component will be able to run without Network. HINT
Summary: Component 'Component43' - ERRORS: 0, WARNINGS: 0, HINTS: 1
---------

---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Usage of Resources in the Execution Environment:

Component Connection 'Request 1':

Component 'Component43', Attribute 'UsedMessageQueue': [UsedMessageQueue(Name=MyMsgQueue, UserName=, UserPwd=, Location=Local,
UsageNecessity=Mandatory, UsageMode=Read, Write, Create, Existence=Checked)]
It is unknown whether message queue exists or not. If message queue exists but cannot be used the component will fail to execute since its usage is mandatory.
- HINT. If message queue does not exist and cannot be created or used the component will fail to execute since its usage is mandatory. - HINT
Summary: Component Connection 'Request 1' - ERRORS: 0, WARNINGS: 0, HINTS: 2
---------

---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the Execution
Environment:

Component Connection 'Request 1':

Method 'Component41.Method1' returns a handle requiring thread affinity which  is piped into an asynchronous method 'Component42.Method1'. The handle
cannot be guaranteed threaf affinity. - ERROR
Summary: Component Connection 'Request 1' - ERRORS: 1, WARNINGS: 0, HINTS: 0
---------
```

Figure 46: Deployment contracts analysis for the Example 14.

For the component C (Component43 in Figure 46), the DCA finds out that although it

requires network through use of a remote XML file which is not available in the assembly's execution environment. the component will be able to run since the usage of the XML file is optional for the component.

Furthermore, the component C requires a message queue. If the queue cannot be created or used, the component will fail to execute.

Moreover, for components A (Component41 in Figure 46) and B (Component42 in Figure 46) the DCA finds out that the return value of the method "A.Method1" is a thread affine handle. The handle is piped into an asynchronous method "B.Method1". Therefore, it cannot be guaranteed thread affinity.

Thus, deployment contracts analysis of the assembly ABC has shown 1 error. Therefore, the assembly ABC is not conflict-free and cannot execute safely at runtime. The component B has to be replaced by another one in the assembly.

5.6.4 Example 15

Consider Example 15 shown in Figure 47. It represents a design pattern for components described in [19, 2]. The design pattern is for systems including one component that loads data in the background and another one that displays the data. Furthermore, while the data is being loaded in the background, the loading component notifies the one displaying the data about the chunks of data already loaded. The component displaying data can either display the chunks of data already loaded, thus implementing so-called streaming, or just display a visualisation of it, e.g. a progress bar, which advances each time the loading component sends a notification that a chunk of data has been loaded.

Figure 47 shows two such components.

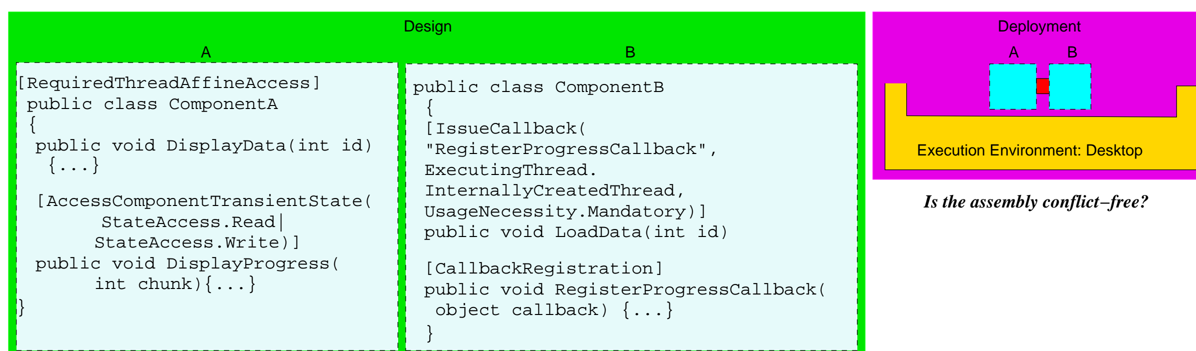


Figure 47: Example 15.

Component A has two methods "DisplayData", which displays loaded data, and "DisplayProgress", which displays a progress bar. A's developer knows that the method "DisplayProgress" may be used as a callback method by another component, which loads the data. They also know that a callback may be invoked on different threads. Since no synchronisation of multiple threads is done inside the component, state corruption will arise if it is used concurrently from multiple threads. Therefore, in the design phase, the component developer is obliged to attach the attribute "RequiredThreadAffineAccess" at component level (in the design phase) to let the system developer know that the component must not be used in multithreaded scenarios.

Component B has two methods: “RegisterProgressCallback” and “LoadData”. The method “RegisterProgressCallback” registers a callback of another component with the component. In this situation, the component developer is obliged to attach the attribute “CallbackRegistration” to the component’s method. The method “LoadData” loads the data. Moreover, while the data is being loaded, the method invokes a callback to notify the component’s user that a certain chunk of data has been loaded. In this situation, the component developer is obliged to attach and parameterise the attribute “IssueCallback”. The attribute parameters show that the method will issue the callback registered with the method “RegisterProgressCallback”. The thread executing the callback will be an internally created one. Furthermore, the callback is mandatory. Therefore, the component must be composed with another component in such a way that the method “RegisterProgressCallback” is called before the method “LoadData” is called.

In the deployment phase, suppose the system developer chooses the desktop as the execution environment.

Furthermore, suppose the system developer decides to compose components A and B in the following way: since A displays the data and needs to know about chunks of data loaded, its method “DisplayProgress” can be registered with B to be invoked as a callback while the data is being loaded by B. Once the data has been loaded, it can be displayed using A’s method “DisplayData”. B offers a method “RegisterProgressCallback” with the attribute “CallbackRegistration” attached. Therefore, this method can be used to register component A’s method “DisplayProgress” as a callback. After that, B’s method “LoadData” can be called to initiate data loading. While the data is being loaded, the method will invoke the registered callback, which is illustrated by the attribute “IssueCallback” attached to the method.

The scenario required by the system developer seems to be fulfilled by assembling components A and B in this way.

To confirm this, we let the DCA check the the assembly AB for conflict-freedom. The analysis result is shown in Figure 48.

```
---------
Check Components' Deployment Contracts For Mutual Compatibility with Respect to Components' Threading and State Models in Consideration of the
Execution Environment:

Component Connection 'Request 1':
Summary: Component Connection 'Request 1' - OK

Component Connection 'Request 2':

Component 'ComponentA' requires thread affine access to one of its parts but can be accessed concurrently by component 'ComponentB'. - ERROR

Component 'ComponentA' accesses its state in not read-only mode. The state can be concurrently accessed by multiple threads from within the component
'ComponentB' but is unprotected from concurrent access by multiple threads. - ERROR
Summary: Component Connection 'Request 2' - ERRORS: 2, WARNINGS: 0, HINTS: 0
---------
```

Figure 48: Deployment contracts analysis for the Example 15.

DCA finds out that component A has a component-level attribute “RequiredThreadAffineAccess” that requires all its methods to be called always from one and the same thread. The method “DisplayProgress” will be called from a thread internally created by the method “LoadData”. But the method “DisplayData” will be called from the main thread. This means that methods of A will be called from different threads, which contradicts its requirement for thread-affine access. Furthermore, if data is loaded several times, the method “B.LoadData(...)” will create a new thread each time it is called thus invoking the method “A.DisplayProgress(...)” each time on a different thread. This means that A and B are incompatible.

A component from the assembly AB has to be replaced by another one. Then a deployment contracts analysis has to be performed again. This process has to be repeated until an assembly of compatible components, i.e. a conflict-free assembly, is found. Once a conflict-free assembly is found, it can be executed at runtime.

6 Evaluation

In this report we have presented conflicts with component assemblies detectable using deployment contracts for components (Section 2). Furthermore, we have shown a way of spotting these conflicts by applying an algorithm presented in Section 3. Moreover, we have proven that the conflicts can be automatically detected by implementing the Deployment Contracts Analyser from Section 4. Finally, we have shown how the DCA can be used for spotting conflicts with component assemblies in Section 5.

The idea of deployment contracts based on a predefined pool of parameterisable attributes can be applied to any component model supporting composition of components at deployment time. We have implemented the idea in .NET, and since the .NET component model supports deployment time composition our implementation is a direct extension of the .NET component model with about 100 new attributes, together with a deployment-time analyser.

Our attributes are created by analysing the APIs of J2EE and .NET frameworks. However, the idea is general and therefore other frameworks for component development can be studied to create more attributes, thus enabling more comprehensive reasoning by extending deployment contracts analysis.

Use of metadata for component deployment in current component models [8] such as EJB and CCM is restricted to component deployment descriptors that are XML specifications describing how to manage components by the component container. Specification of metadata in an easily changeable form like XML has the disadvantage that it can be easily tampered with, which may be fatal for system execution. Therefore, our metadata is contained in the real binary components, cannot be easily tampered with and is retrieved automatically by the Deployment Contracts Analyser.

Moreover, metadata about components in deployment descriptors is not analysed for component mutual compatibility. Although deployment descriptors allow specification of some environmental dependencies and some aspects of threading, the information specifiable there is not comprehensive and only reflects features that are manageable by containers, which are limited. By contrast, our metadata set is comprehensive and the component developer is obliged to show all environmental dependencies and aspects of threading for their component. In addition, our deployment contracts analysis takes account of properties of the system execution environment, as well as emergent assembly-specific properties like e.g. transient state, which other approaches do not do.

Furthermore, in current component models employing metadata for component deployment, metadata is not analysed at deployment time. For instance, in EJB and CCM the data in deployment descriptors is used by containers at runtime but not at deployment time. The deployment descriptor has to be produced at deployment time but its contents are used at runtime. In .NET, only metadata for graphical component arrangement is analysed at deployment time. By contrast, in our approach all the metadata is analysed at deployment time, which is essential when components come from different suppliers.

Currently the J2EE and .NET frameworks provide compilers for their components. How-

ever, if components are produced and compiled independently by component developers and composed later in binary form by system developers, no means for compiler-like checking of composition is provided. By contrast, our Deployment Contracts Analyser can check components for compatibility when they are in binary form and ready to be composed by a compositon operator.

Implementation done consists of roughly 30.000 net lines of C# code (LOC). Deployment Contracts comprise 10.000 LOC, Deployment Contracts Analyser – 15.000 LOC, and test components – 5.000 LOC.

Moreover, our metadata attributes may require different analysis in different component models. For instance, on the one hand attributes related to callbacks may require an extensive analysis in component models where callback are allowed. On the other hand, the same attributes may mean that components are just unsuitable in component models where callbacks are not allowed. For instance, in .NET component model, callbacks are allowed, whereas in component model with exogenous connectors not. DCA allows the system developer to see deployment contracts of components prior to deployment contracts analysis. Therefore, the system developer may preselect component by themselves.

Finally, system deployment is a known research area today. However, component deployment is a new, exciting, research area. The work in this report makes a contribution to this area.

7 Conclusion

In this report we have shown how component composition can be established by checking deployment contracts of components. We have also shown that the process of deployment contracts analysis can be widely automated. The results of deployment contracts analysis for a component assembly are meant to be reviewed by the system developer, who has the knowledge of the overall system and its execution environment, and can therefore decide whether certain findings can be ignored or, conversely, represent serious errors for the system.

We have also shown examples of detected conflicts, which are ignored by current component models that do not have and check deployment contracts of components.

Overall, checking component deployment contracts on component deployment helps make software systems more reliable since conflicts with component assemblies can be discovered at deployment time before runtime, and the assembly at deployment time can still be changed to ensure it is conflict-free before going into the runtime phase.

8 Appendix

8.1 Code outline for checking mutual compatibility of deployment contracts of components with respect to usage of resources in execution environment

```
...
#region Analysis of use of a resource by several components

IList anAlreadyCheckedAttributesList = new ArrayList();

// iterate through all components in the current request
anOuterIterationsCounter = 0;
IList aFoundAttributesList = new ArrayList();
```

```
foreach(ComponentShell aComponentShell in GetComponentsFromRequest(aRequest.RequestDescription))
{
    anOuterIterationsCounter++;

    // iterate through all the attributes checked here
    foreach(string anAttributeName in myAttributeListForAnalysis1)
    {
        if (anAlreadyCheckedAttributesList.Contains(anAttributeName))
        {
            continue;
        }

        Attribute anAttribute = null;

        // Is there the current attribute on connected method
        // or component?
        if (((anAttribute = ExistsAttributeOnComponentMethod(
            RetrieveComponentByName(aComponentShell.ComponentName),
            aComponentShell.MethodName,
            anAttributeName) as Attribute) != null)
            ||
            ((anAttribute = ExistsAttributeOnComponent(
            RetrieveComponentByName(aComponentShell.ComponentName),
            anAttributeName) as Attribute) != null))
        {
            anAlreadyCheckedAttributesList.Add(anAttributeName);

            UsageModeObject aUsageModeObject = GetUsageMode(anAttribute);
            UsageNecessityObject aUsageNecessityObject = GetUsageNecessity(anAttribute);
            ExistenceObject anExistenceObject = GetExistence(anAttribute);
            string aRepresentedResource = GetRepresentedResource(anAttribute);

            // check only those with UsageMode parameter
            if (aUsageModeObject != null)
            {
                // check if components in the chain have the
                // same attribute at component or
                // connected method level
                int anInnerIterationsCounter = 0;
                Attribute anAttribute1 = null;
                ResourceState aRRState = new ResourceState();
                aRRState.Unknown = true;
                UsageModeObject aUsageModeObjectPrev = new UsageModeObject();
                aUsageModeObjectPrev = aUsageModeObject;
                IList aTrustedConnectionList = new ArrayList();

                anAnalysisResult +=
                    CheckComponentWithRespectToExistenceAndCreation(
                    "\n" + "Component '" + aComponentShell.ComponentName + "', Attribute '"
                    + anAttribute.GetType().Name + "': "
                    + AttributeHandlingUtils.ExpandAttribute(anAttribute, false),
                    aRepresentedResource,
                    aRRState,
                    aUsageModeObject,
                    aUsageNecessityObject,
                    anExistenceObject);

                SetTrustedConnection(ref aTrustedConnectionList, anAttribute);

                foreach(ComponentShell aComponentShell1 in
                   GetComponentsFromRequest(aRequest.RequestDescription))
                {
                    anInnerIterationsCounter++;

                    if(anInnerIterationsCounter > anOuterIterationsCounter)
                    {
                        // is there the current attribute on connected method
```

```
// or component
if (((anAttribute1 = ExistsAttributeOnComponentMethod(
    RetrieveComponentByName(aComponentShell1.ComponentName),
    aComponentShell1.MethodName,
    anAttributeName) as Attribute) != null)
    ||
    ((anAttribute1 = ExistsAttributeOnComponent(
    RetrieveComponentByName(aComponentShell1.ComponentName),
    anAttributeName) as Attribute) != null))
{
    UsageModeObject aUsageModeObject1 = GetUsageMode(anAttribute1);
    UsageNecessityObject aUsageNecessityObject1 =
       GetUsageNecessity(anAttribute1);
    ExistenceObject anExistenceObject1 = GetExistence(anAttribute1);
    string aRepresentedResource1 = GetRepresentedResource(anAttribute1);

    // check only attributes with with UsageMode parameter
    // and pointing to the same resource
    if ((aUsageModeObject1 != null)
        &&
        (IsSameResource(anAttribute, anAttribute1)))
    {
        // now compare aUsageModeObjectPrev and aUsageModeObject1
        anAnalysisResult += ChangeRRState(ref aRRState,
           aUsageModeObjectPrev, aUsageModeObject1);

        anAnalysisResult +=
            CheckComponentWithRespectToExistenceAndCreation(
            "\n" + "Component '" + aComponentShell1.ComponentName
            + "', Attribute '"
            + anAttribute1.GetType().Name + "': "
            + AttributeHandlingUtils.ExpandAttribute(anAttribute1, false),
            aRepresentedResource1,
            aRRState,
            aUsageModeObject1,
            aUsageNecessityObject1,
            anExistenceObject1);

        anAnalysisResult += ChangeRRState(ref aRRState,
           aUsageModeObject1, null);

        aUsageModeObjectPrev = aUsageModeObject1;

        SetTrustedConnection(ref aTrustedConnectionList, anAttribute1);

        // is the current component the last one in the chain
        // and is the connection recurrent?
        if ( (anInnerIterationsCounter ==
           GetComponentsFromRequest(aRequest.RequestDescription).Count)
            &&
            (aRequest.Recurrent))
        {
            anAnalysisResult +=
                CheckComponentWithRespectToExistenceAndCreation(
                "\n" + "Component '" + aComponentShell.ComponentName
                + "', Attribute '"
                + anAttribute.GetType().Name + "': "
                + AttributeHandlingUtils.ExpandAttribute(
                   anAttribute1, false),
                aRepresentedResource,
                aRRState,
                aUsageModeObject,
                aUsageNecessityObject,
                anExistenceObject);
        }
    }
}
```

```
                }
            }

            anAnalysisResult += CheckTrustedConnection(aTrustedConnectionList);
        }
    }
  }
}

#endregion
...
```

8.2 Code outline for checking mutual compatibility of deployment contracts of components with respect to their threading models in consideration of state and concurrency management of execution environment

```
...
#region Check components' threading models in desktop and web execution environment

if (Project.ExecutionEnvironment_.Type == ExecutionEnvironmentType.Desktop)
{
    if (aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
    {
        int anOuterIterationsCounter = 0;
        foreach(ComponentShell aComponentShell in GetComponentsFromRequest(aRequest.RequestDescription))
        {
            anOuterIterationsCounter++;

            if (aComponentShell.ComponentThreadingModel ==
                ComponentThreadingModel.MultithreadedComponent)
            {
                int anInnerIterationsCounter = 0;
                foreach(ComponentShell aComponentShell1 in
                        GetComponentsFromRequest(aRequest.RequestDescription))
                {
                    anInnerIterationsCounter++;

                    if(anInnerIterationsCounter > anOuterIterationsCounter)
                    {
                        object aSpawnThread = ExistsAttributeAtAnyCompLevel(
                            RetrieveComponentByName(aComponentShell.ComponentName),
                            "SpawnThread");

                        object anAsynchronousMethod = ExistsAttributeAtAnyCompLevel(
                            RetrieveComponentByName(aComponentShell.ComponentName),
                            "AsynchronousMethod");

                        IssueCallback anIssueCallback = ExistsAttributeAtAnyCompLevel(
                            RetrieveComponentByName(aComponentShell.ComponentName),
                            "IssueCallback") as IssueCallback;

                        if ((anAsynchronousMethod != null)
                            ||
                            (anIssueCallback != null))
                        {
                            if (aRequest.Recurrent)
                            {
                                anAnalysisResult += CheckThreadAffineElements(aComponentShell1,
                                    aComponentShell, "ERROR");

                                if (aRequest.AssemblyStateModel !=
                                    AssemblyStateModel.StatelessAssembly)
                                {
                                    anAnalysisResult += CheckStateUsage(aComponentShell1,
                                        aComponentShell, "ERROR");
```

```
                        }

                        anAnalysisResult += CheckSingletonUsage(aComponentShell1,
                            aComponentShell, "ERROR");

                        anAnalysisResult += CheckStaticVariableUsage(aComponentShell1,
                            aComponentShell, "ERROR");
                    }
                }
                else if (aSpawnThread != null)
                {
                    anAnalysisResult += CheckThreadAffineElements(aComponentShell1,
                        aComponentShell, "WARNING");

                    if (aRequest.AssemblyStateModel != AssemblyStateModel.StatelessAssembly)
                    {
                        anAnalysisResult += CheckStateUsage(aComponentShell1,
                            aComponentShell, "WARNING");
                    }

                    anAnalysisResult += CheckSingletonUsage(aComponentShell1,
                        aComponentShell, "WARNING");

                    anAnalysisResult += CheckStaticVariableUsage(aComponentShell1,
                        aComponentShell, "WARNING");
                }
            }
        }
    }
  }
 }
}
else if (Project.ExecutionEnvironment_.Type == ExecutionEnvironmentType.Web)
{
    if (Project.ExecutionEnvironment_.Properties.SystemInstantiation_ == SystemInstantiation.Unknown)
    {
        anAnalysisResult +=
        "Assembly instantiation mode is not specified for web environment. Please, choose one. - ERROR";
        anAnalysisResult += "\n";
    }
    else if (Project.ExecutionEnvironment_.Properties.SystemInstantiation_ ==
             SystemInstantiation.OncePerRequest)
    {
        string anAssemblyInstantiationMode = "assembly instance per request";

        // singlethreaded, stateless assembly -> no analysis needed in this case

        // singlethreaded, stateful assembly
        if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.SinglethreadedAssembly)
            &&
            (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
        {
            anAnalysisResult += CheckStateStorageUsage(aRequest.RequestDescription,
                anAssemblyInstantiationMode);
        }
            // multithreaded, stateless assembly
        else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
            &&
            (aRequest.AssemblyStateModel == AssemblyStateModel.StatelessAssembly))
        {
            anAnalysisResult += CheckThreadingMultithrAsmEnvAsmInstPerReq(false,
                aRequest.RequestDescription);
        }
            // multithreaded, stateful assembly
        else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
            &&
            (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
```

```
    {
        anAnalysisResult += CheckThreadingMultithrAsmEnvAsmInstPerReq(true,
            aRequest.RequestDescription);

        anAnalysisResult += CheckStateStorageUsage(aRequest.RequestDescription,
            anAssemblyInstantiationMode);
    }
}
else if (Project.ExecutionEnvironment_.Properties.SystemInstantiation_ ==
         SystemInstantiation.OncePerUserSession)
{
    string anAssemblyInstantiationMode = "assembly instance per user session";
    // singlethreaded, stateless assembly
    if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.SinglethreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatelessAssembly))
    {
        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);
    }
        // singlethreaded, stateful assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.SinglethreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
    {
        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);

        anAnalysisResult += "The assembly is deployed into the web environment with "
            + "assembly instantiation mode '" + anAssemblyInstantiationMode
            + "'. State is only retained for a user session. Check if it is appropriate. - WARNING";
        anAnalysisResult += "\n";
    }
        // multithreaded, stateless assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatelessAssembly))
    {
        anAnalysisResult += CheckThreadingModelsOfMultithrAsmsInWebEnv(
            aRequest.RequestDescription, false, true, aRequest.Recurrent);

        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);
    }
        // multithreaded, stateful assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
    {
        anAnalysisResult += CheckThreadingModelsOfMultithrAsmsInWebEnv(
            aRequest.RequestDescription, true, true, aRequest.Recurrent);

        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);

        anAnalysisResult += "The assembly is deployed into the web environment with "
            + "assembly instantiation mode '" + anAssemblyInstantiationMode
            + "'. State is only retained for a user session. Check if it is appropriate. - WARNING";
    }
}
else if (Project.ExecutionEnvironment_.Properties.SystemInstantiation_ ==
         SystemInstantiation.OnceForAllConcurrentRequests)
{
    string anAssemblyInstantiationMode = "assembly instance for all requests";

    // singlethreaded, stateless assembly
    if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.SinglethreadedAssembly)
```

```
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatelessAssembly))
    {
        anAnalysisResult += CheckThreadingModelsOfSinglethAsmsAsmInstForAllReqs(
            aRequest.RequestDescription, false);
    }
        // singlethreaded, stateful assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.SinglethreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
    {
        anAnalysisResult += CheckThreadingModelsOfSinglethAsmsAsmInstForAllReqs(
            aRequest.RequestDescription, true);
    }
        // multithreaded, stateless assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatelessAssembly))
    {
        anAnalysisResult += CheckThreadingModelsOfMultithrAsmsInWebEnv(
            aRequest.RequestDescription, false, false, aRequest.Recurrent);

        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);
    }
        // multithreaded, stateful assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
    {
        anAnalysisResult += CheckThreadingModelsOfMultithrAsmsInWebEnv(
            aRequest.RequestDescription, true, false, aRequest.Recurrent);

        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);
    }
}
else if (Project.ExecutionEnvironment_.Properties.SystemInstantiation_ ==
         SystemInstantiation.PoolSynchronisedInstancesForAllConcurrentRequests)
{
    string anAssemblyInstantiationMode = "pool of synchronised assembly instances for all requests";

    // singlethreaded, stateless assembly
    if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.SinglethreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatelessAssembly))
    {
        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);
    }
        // singlethreaded, stateful assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.SinglethreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
    {
        anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
            aRequest.RequestDescription, anAssemblyInstantiationMode);

        anAnalysisResult += CheckStateStorageUsage(aRequest.RequestDescription,
            anAssemblyInstantiationMode);
    }
        // multithreaded, stateless assembly
    else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
        &&
        (aRequest.AssemblyStateModel == AssemblyStateModel.StatelessAssembly))
    {
        anAnalysisResult += CheckThreadingModelsOfMultithrAsmsInWebEnv(
```

```
                aRequest.RequestDescription, false, true, aRequest.Recurrent);

            anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
                aRequest.RequestDescription, anAssemblyInstantiationMode);
        }
            // multithreaded, stateful assembly
        else if ((aRequest.AssemblyThreadingModel == AssemblyThreadingModel.MultithreadedAssembly)
            &&
            (aRequest.AssemblyStateModel == AssemblyStateModel.StatefulAssembly))
        {
            anAnalysisResult += CheckThreadingModelsOfMultithrAsmsInWebEnv(
                aRequest.RequestDescription, true, true, aRequest.Recurrent);

            anAnalysisResult += CheckThreadAffinityForAllCompsInWebEnv(
                aRequest.RequestDescription, anAssemblyInstantiationMode);

            anAnalysisResult += CheckStateStorageUsage(aRequest.RequestDescription,
                anAssemblyInstantiationMode);
        }
    }
}

#endregion
...
```

References

[1] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical Report CMU/SEI-2000-TR-008, Carnegie Melon Software Engineering Institute, 2000.

[2] Microsoft Corporation. Microsoft Asynchronous Pattern for Components.

[3] Microsoft Corporation. MSDN – .NET Framework Class Library, Version 2.0, 2005.

[4] R. Englander. *Developing Java Beans*. O'Reilly & Associates, 1997.

[5] M. Fowler, D. Box, A. Hejlsberg, A. Knight, R. High, and J. Crupi. The great J2EE vs. Microsoft .NET shootout. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 143–144, New York, NY, USA, 2004. ACM Press.

[6] A. W. Keen and R. A. Olsson. Exception handling during asynchronous method invocation. In *Parallel Processing: 8th International Euro-Par Conference Paderborn, Germany*, volume 2400 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[7] K.-K. Lau and V. Ukis. A Container for Automatic System Control Flow Generation using Exogenous Connectors. Preprint 31, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, August 2005. ISSN 1361 - 6161.

[8] K.-K. Lau and V. Ukis. Component Metadata in Component-based Software Development: A Survey. Preprint 34, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, October 2005. ISSN 1361 - 6161.

[9] K.-K. Lau and V. Ukis. Automatic Control Flow Generation from Software Architectures. In *Proceedings of the 5th International Symposium on Software Composition*, volume 4089 of *LNCS*, pages 325–339, Vienna, Austria, March 2006. Springer.

[10] K.-K. Lau and V. Ukis. Defining and Checking Deployment Contracts for Software Components. In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, volume 4063 of *LNCS*, pages 1–16, Stockholm, Sweden, June 2006. Springer.

[11] K.-K. Lau and V. Ukis. Deployment Contracts for Software Components. Preprint 36, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, February 2006. ISSN 1361 - 6161.

[12] K.-K. Lau, V. Ukis, P. Velasco, and Z. Wang. A Component Model for Separation of Control Flow from Computation in Component–Based Systems. In *Proceedings of the 1st International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems, Elsevier ENTCS*, Nuremberg, Germany, November 2005.

[13] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. SIGSOFT Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106, 2005.

[14] K.-K. Lau and Z. Wang. A survey of software component models. Preprint CSPP-30, School of Computer Science, The University of Manchester, April 2005.

[15] V. Matena and B. Stearns. *Applying Enterprise JavaBeans – Component-based Development for the J2EE Platform.* Addison-Wesley, 2000.

[16] Microsoft .NET web page. `http://www.microsoft.com/net`.

[17] D. S. Platt. *Introducing Microsoft .NET.* Microsoft Press, 3rd edition, 2003.

[18] D. C. Schmidt, T. Harrison, and N. Pryce. Thread-specific storage - an object behavioral pattern for accessing per-thread state efficiently. In *The Pattern Languages of Programming Conference*, September 1997.

[19] Douglas C. Schmidt. *Pattern-oriented Software Architecture. Vol. 2, Patterns for Concurrent and Networked Objects.* New York John Wiley&Sons, Ltd., 2000.

[20] Sun Microsystems. *Java 2 Platform, Enterprise Edition.* `http://java.sun.com/j2ee`.

[21] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation.* Prentice Hall, second edition, 1997.

[22] A. Wigley, M. Sutton, R. MacLeod, R. Burbidge, and S. Wheelwright. *Microsoft .NET Compact Framework(Core Reference).* Microsoft Press, January 2003.