Defining and Checking Deployment Contracts for Software Components

Kung-Kiu Lau and Vladyslav Ukis

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

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

1 Introduction

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

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

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

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

I. Gorton et al. (Eds.): CBSE 2006, LNCS 4063, pp. 1–16, 2006.
© Springer-Verlag Berlin Heidelberg 2006

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

2 Component Deployment

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

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

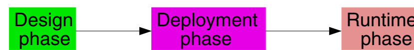

Fig. 1. Software component lifecycle

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

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

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

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

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

2.1 Current Component Models

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

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

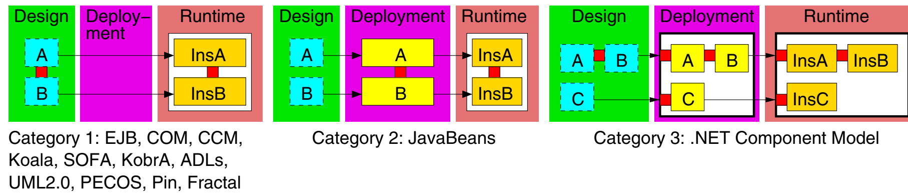

Fig. 2. Current component models

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

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

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

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

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

2.2 Composition in the Deployment Phase

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

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

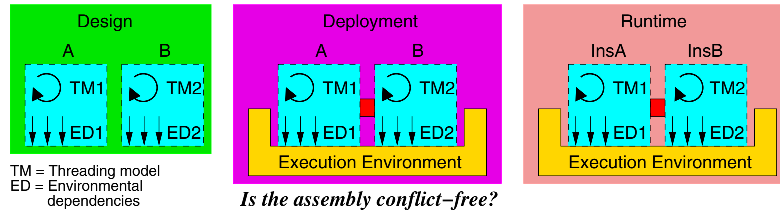

Fig. 3. Composition in deployment phase

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

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

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

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

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

2.3 Deployment Contracts

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

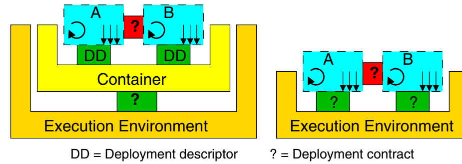

Fig. 4. Deployment contracts

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

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

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

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

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

3 Defining Deployment Contracts

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

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

3.1 Environmental Dependencies

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

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

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

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

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

Fig. 5. A component with an environmental dependency

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

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

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

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

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

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

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

3.2 Threading Models

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

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

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

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

Fig. 6. A component with a defined threading model

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

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

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

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

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

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

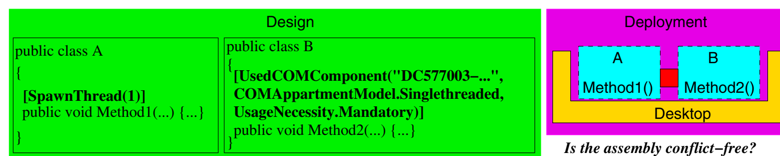

Fig. 7. Example 1 using metadata attributes

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

[5] In .NET Framework v2.0 there are about 200 attributes, but they are only checked at runtime.
[6] Note that this problem may also arise in other component models.

on an internally created thread. Therefore, if method "A.Method1()" is invoked several times, each time a new thread is created that makes an invocation of the method "B.Method2()". Therefore, the COM component used by method "B.Method2()" is not going to be called from one thread and its requirement for a single threaded apartment cannot be fulfiled in such composition of components A and B. Therefore, the system developer can be warned not to do such composition.

Besides this, using a COM component requires use of a file system, where the component resides, and Windows Registry, where it must be registered. The system developer can also be warned if these resources are unavailable in the system execution environment.

Moreover, in Fig. 7 the components are deployed into the desktop environment. In this environment, there is a guarantee that the main thread of the system is always the same for the lifetime of a system instance. Therefore, the system developer need not be warned that the execution environment may cause problems. Note that in the web environment there is no guarantee for the thread affinity of the main thread. If the assembly in Fig. 7 was deployed into the web environment, it would also fail since the COM component used by the component B would be accessed by different threads imposed by the web environment.

3.3 Implementing Retrievable Metadata

The attributes we have created must be retrievable at deployment time, i.e. they must be retrievable from binaries. In this section, we explain how we implement them.

Our implementation draws on .NET's facility for defining custom attributes[7]. A custom attribute in .NET is a class derived from the .NET's class System.Attribute. An example of an attribute from the attribute pool we have defined is shown below:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Method|
                AttributeTargets.Property, AllowMultiple=true)]
public class UsedWebService : System.Attribute {
  public UsedWebService(string url, string userName,
    string pwd, UsageNecessity usageNecessity) {...} ... }
```

The attribute above is called 'UsedWebService'. It has a constructor, which takes as parameters the url to the web service, credentials used when accessing the web service as well as whether the web service usage is mandatory for the component.

Furthermore, above the attribute declaration 'public class UsedWebService : System.Attribute', the usage of the attribute is specified by a .NET built-in attribute 'AttributeUsage' that indicates which elements of components the attribute is allowed to be applied to, as well as whether multiple attributes can be applied to the same element. Here the attribute 'UsedWebService' can be applied to either a whole class (we model components as classes) or a component's method or property. Here 'AllowMultiple=true' means that the attribute 'UsedWebService' can be applied multiple times to the same component element. That is, if a component makes use of several web services, several 'UsedWebService' attributes can be applied to indicate the component's environmental dependencies.

[7] In Java, Annotations can be used to express the metadata. However, they are somewhat less flexible than .NET Attributes.

To retrieve attributes from a binary component, we use .NET's Reflection facility from System.Reflection namespace. For instance, to retrieve attributes at component level, the following code is executed:

```
Type compType = Type.GetType(componentName);              (i)
object[] attributes = compType.GetCustomAttributes(false);(ii)
```

(i) loads the component type from the binary component using component name in a special format, and (ii) retrieves all the attributes attached to the component. Note that no component instantiation has been done.

To retrieve attributes on component's properties, the following code is executed:

```
Type compType = Type.GetType(componentName);              (i)
foreach(PropertyInfo prop in compType.GetProperties())   (ii)
{object[] attributes = prop.GetCustomAttributes(false);} (iii)
```

(i) loads the component type from the binary component, (ii) iterates through all the properties inside the component, and (iii) retrieves all the attributes attached to the current property.

Attributes attached to component's methods, method's parameters and return values can be retrieved in a similar but more complicated manner.

Being able to retrieve the attributes at deployment time enables us to check deployment contracts before component instantiation at run time.

4 Checking Deployment Contracts

Given an assembly of components with deployment contracts and a chosen execution environment in the deployment phase, as illustrated by Fig. 4, we can use the deployment contracts to determine whether the assembly is conflict-free. In this section we explain how we do so.[8]

The checking process first loads the binary components, and then for each binary retrieves the attributes at all levels (component, property, method, and method input and return parameters). The checking task is then divided into 2 sub-tasks: (i) Analysis of mutual compatibility of deployment contracts of components in the assembly with respect to usage of resources in the assembly's execution environment; (ii) 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. Both sub-tasks consist of checking the deployment contracts involved. The results of the checking range over {ERROR, WARNING, HINT} with the obvious meaning.

For (i), we perform the following: For each attribute at any level we determine resource(s) required in the execution environment. If a resource is not available in the execution environment, an ERROR is issued.

Furthermore, we follow component connections in the assembly and consider how resources are used by the individual components by evaluating attached attributes'

[8] We present only an outline here.

parameters. Once an attribute representing a resource usage is found on a component, we follow the chain of components till another component with an attribute representing the usage of the same resource is found either at method or property or component level. Once such a component is found, we check the "UsageMode" parameters of the attributes on the two components for compatibility and issue ERROR, WARNING or HINT depending on the parameters' values. After that, we again follow the chain of components till the next component with an attribute representing the usage of the same resource is found and check the values of the parameter "UsageMode" on corresponding attributes of the component and the previous one in the chain. This process is repeated till all attributes representing resource usage on all components are processed.

Moreover, specific parameters of each attribute are analysed and WARNINGs and HINTs are issued if necessary. For instance, if attributes' parameters indicate that components in a component assembly use a database and not all components uniformly use either encrypted or unencrypted database connection, a WARNING is issued.

Another example is usage of cryptography files. If a cryptography file is used, it is hinted which cryptography algorithm has been used to create the certificate. This information is useful to the system developer due to the fact the different cryptography algorithms have different degrees of security and different processing times when checked. Depending on system requirements a specific cryptography algorithm may or may not be suitable.

A further example is represented by communication channels. 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, who knows system requirements, to judge whether the component is suitable for their system.

For (ii), we perform the following: We follow component connections in the assembly to determine for each component if it is stateful or stateless, and multithreaded or singlethreaded. This can be done by evaluating corresponding attributes on a component. After that we determine if the assembly is stateful or stateless, and multithreaded and singlethreaded depending on the components in the assembly. If at least one component in the assembly is stateful, the assembly is stateful. Otherwise, it is stateless. If at least one component in the assembly is multithreaded, the assembly is multithreaded. Otherwise, it is singlethreaded.

Following this, we check whether state management of the assembly's execution environment is suitable for the assembly. Furthermore, we check whether concurrency management of the assembly's execution environment is suitable for the assembly. We issue ERRORs, WARNINGs or HINTs depending on the level of incompatibility.

Apart from that, if a component can repeatedly issue a callback to another one on an internally created thread, and the callback method either requires thread-affine access; or accesses component's transient state in not read-only mode, or accesses a singleton or a static variable, and no component element enclosing it is marked as reentrant or thread-safe, an ERROR is issued pointing out a likely state corruption problem.

Moreover, if synchronisation primitives are exchanged between components, a WARNING is issued pointing out a possible cause for a deadlock.

5 Example

To illustrate the usefulness of deployment contracts we show how they can be applied to a design pattern described in [4, 5]. 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.

Fig. 8 shows two such components. 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.

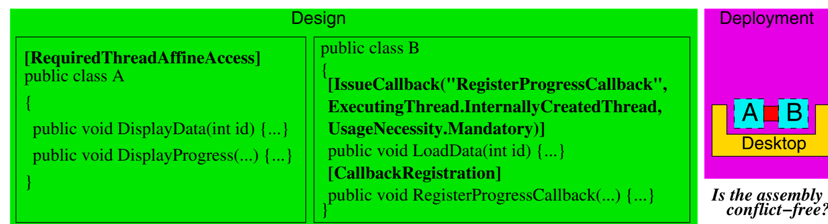

Fig. 8. Implementation of a design pattern for components with use of metadata attributes

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, he can check the deployment contracts of A and B in the manner described in the previous section. We have implemented a Deployment Contracts Analyser (DCA) for automating the checking process. For this example, the result given by DCA is shown Fig. 9.

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.

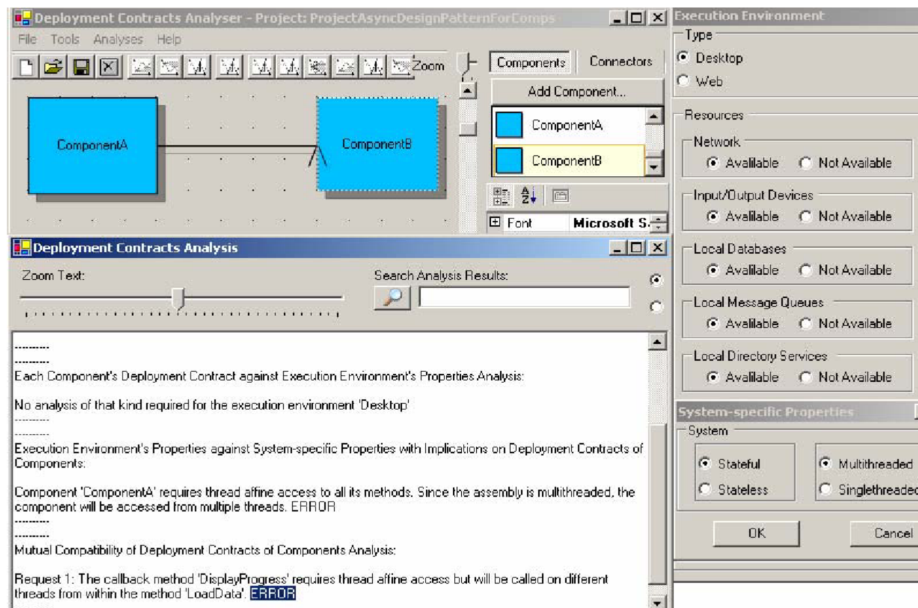

Fig. 9. Deployment Contracts Analyser

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

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 (Fig. 2), 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.

Our pool of metadata for component deployment is general-purpose since it is created by analysing general-purpose frameworks. Other pools of metadata for component deployment, see [12] for a survey, are mostly not general-purpose. For example, MetaH has a set of metadata for the domains of flight control and avionics; the CR-RIO Framework has metadata for distribution and processing policies.

Use of metadata for component deployment in current component models [12] 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. However, 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.

Using our attributes, developers have extensive IDE support in the form of IntelliSense. Moreover, .NET developers should be familiar with the concept of attributes thus making it easy for them to employ the proposed approach using new attributes. Thanks to various parameters on each attribute, the component developer can flexibly specify how resources are used inside components and which threading aspects are available.

Furthermore, although EJB specification forbids component developers to manage threads themselves, there is nothing in current EJB implementations that would prevent the developers to do so. If enterprise beans manage threads themselves, they may interfere with the EJB container and cause the running system to fail. By contrast, our approach checks threading models of components for compatibility before runtime, thus enabling the system developer to recognise and prevent runtime conflicts before runtime.

7 Conclusion

In this paper,we have shown how to use metadata to define deployment contracts of components that express component's environmental dependencies and threading model. Such contracts bind two parties: (a) the component developer, who develops components, and (b) the system developer, who develops systems by composing pre-existing components using composition operators. The former is obliged to attach the attributes to component's elements in specified cases. The latter is guaranteed to be shown conflicts among the third-party components in assemblies they create at deployment time.

We have also shown how deployment contracts analysis can be performed to help the system developer spot these conflicts. Most importantly, incompatible components in an assembly can be replaced by other, compatible, ones to ensure conflict-freedom of the assembly, before runtime.

Besides checking deployment contracts at deployment time, we have also implemented a generic container for automated binary component composition [13] using special composition operators – exogenous connectors [15]. Our future work will combine the generic container and the Deployment Contracts Analyser, thus allowing automated component composition only if the analyser does not discover any conflicts with component assembly.

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. B. Boehm and C. Abts. COTS integration: Plug and pray? *IEEE Computer*, 32(1):135–138, 1999.
3. D. Box. *Essential COM*. Addison-Wesley, 1998.
4. Schmidt D. C. *Pattern-oriented Software Architecture. Vol. 2, Patterns for Concurrent and Networked Objects*. New York John Wiley&Sons, Ltd., 2000.
5. Microsoft Corporation. Microsoft asynchronous pattern for components.
6. Microsoft Corporation. Msdn .net framework class library version 2.0, 2005.
7. R. Englander. *Developing Java Beans*. O'Reilly & Associates, 1997.
8. EZ JCom Framework web page. http://www.ezjcom.com.
9. 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.
10. G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. K.-K. Lau and Z. Wang. A taxonomy of software component models. In *Proc. 31st Euromicro Conference*. IEEE Computer Society Press, 2005.
17. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *Proc. ACM SIGSOFT'96*, pages 24–32, 1996.
18. Sun Microsystems. Enterprise java beans specification, version 3.0, 2005.
19. Object Management Group (OMG). Corba components, specification, version 0.9.0, 2005.
20. D.S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 3rd edition, 2003.
21. 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.
22. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
23. Sun Microsystems. *Java 2 Platform, Enterprise Edition*. `http://java.sun.com/j2ee/`.
24. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
25. A. Wigley, M. Sutton, R. MacLeod, R. Burbidge, and S. Wheelwright. *Microsoft .NET Compact Framework(Core Reference)*. Microsoft Press, January 2003.