October 2005

Component Metadata in Component-based Software Development: A Survey

Kung-Kiu Lau and Vladyslav Ukis

School of Computer Science
The University of Manchester
Preprint Series
CSPP-34

Component Metadata in Component-based Software Development: A Survey

Kung-Kiu Lau and Vladyslav Ukis

October 2005

Abstract

We present a survey of systems utilizing component metadata for different tasks. Subsequently, we identify key properties of metadata usage and align the systems according to that properties. The survey reveals areas in Component-Based Software Development (CBSD) still untackled and open for further research.

Keywords: software components, component models, component metadata

Contents

List of Figures

List of Tables

1 Introduction

Component-based Software Development (CBSD) is an evolving area of Computer Science with many facets that are being investigated by researchers and practitioners around the world. This report is devoted to an area of CBSD, which is rather immature yet. We investigate current developments in the area of component metadata. First, we provide a survey of component models, ADLs and related systems exploiting component metadata for different tasks. Subsequently, we make a summary and show that there is room for further research towards component metadata especially in the context of component models.

2 Software Components

In this section we define what we understand by the term software component as well as define software component's lifecycle phases.

2.1 Definition

The term Software Component is not uniformly defined in Component-Based Software Development (CBSD) and has been used in the literature with various meanings. Therefore it is essential to clearly state what is understood by Software Component in this work before proceeding. There are several currently most adopted definitions of software component. Among them there are two, a mixture of which this work will adopt. The first definition is given by Szyperski [24] and is the following:

```
"A software component is a unit of composition with contractually
specified interfaces and explicit context dependencies only. A
software component can be deployed independently and is subject to
composition by third parties"
```

The second one is given by Heineman and Council [24] and states that

```
"A [component is a] software element that conforms to a component
model and can be independently deployed and composed without
modification according to a composition standard."
```

We think that on the one hand the definition given by Szyperski does not require that a component[1] must conform to a component model, and on the other hand the definition proposed by Heineman and Council does not mention that a components interface is a contract between the component and its clients. Both definitions adopt the fact that a component is unit of independent deployment, which we fully agree with.

2.2 Lifecycle Phases of Software Components

Having clarified what we understand by the term 'component' we go on to describe what we think to be phases in a component's lifecycle, as these are also not uniformly defined in CBSD. We need to define the phases here as we need them during our analysis of component metadata usage.

[1]We mean 'software component' when referring to the term 'component' in this work.

We identify 6 phases a component can go through during its lifecycle. These are Design Phase, Compilation Phase, Deployment Phase, Runtime Phase, Update Phase and Removal Phase. The relationships between the phases are depicted in Figure 1 and explained in the following.

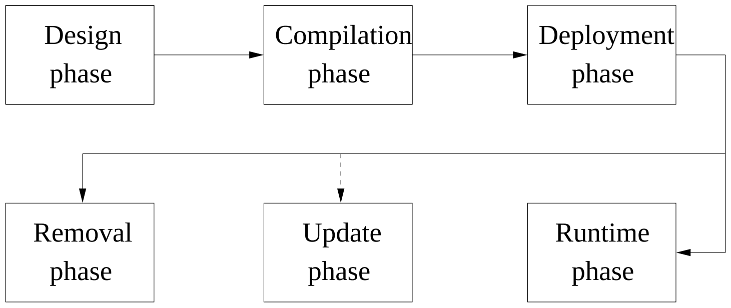


Figure 1: Component lifecycle phases

Component design phase First of all, a component must be designed. This includes all activities, which lead to component's source code in a programming language ready to be compiled. This phase is referred to as 'design phase' of component. During design phase a component can be specified using some specification (e.g. UML or an ADL specification), which will ultimately be compiled into source code. So, the input of design phase of component is an idea about what component is supposed to do and the output is complete source code for the component. During design phase composite components can be built from component templates. But the outcome remains the same. At the end of the component design phase, a component is designed and implemented, and its source code is available and ready to be compiled.

Component compilation phase This phase takes as input the source code from the design phase and compiles the component into a binary unit using a compiler for the programming language the component is written in. This step is completely automated. Component developer is not involved in compilation further than to start it off. The outcome of the component compilation phase is a binary component, which is ready to be deployed. Binary components are ready to be sold as they, unlike the source code from component design phase, do not reveal intellectual property about component implementation in a human readable form.

Component deployment phase In this phase a binary component can be either integrated into an application or become part of a composite component (in fact a component is intended to be integrated into as many applications or composite components as possible to foster reuse). The application itself the component is being integrated into is in its design phase, but the component is in deployment phase. This is an important fact, which often causes confusion of terminology. The same applies when a binary component is composed with another component to build a composite component. The composite component is being designed and making use of the binary component, which is in deployment phase.

An important distinction between component design and deployment phase is that at design phase component's code is designed, developed and changed, whereas at deployment phase the code is not changed any more and the component is in binary form ready for further integration or composition. It is worth stressing that the system the component is being integrated to is in its design phase whereas the component itself is already in deployment phase.

At deployment phase, various tasks may be necessary to prepare the component to run. For instance, a component may need to be registered with a naming service to be looked up at runtime. Some component models employ containers for hosting components. Components are required to be provided with some descriptors to allow containers to instantiate them at runtime.

To summarise, the input of component deployment phase is a binary component and the output is the binary component prepared to execute in the target environment. I.e. the component is integrated into the target environment be it an application or a composite component and is registered with some system entities like naming services or containers if necessary.

Component runtime phase At component runtime phase a component instance is created and running. The input for the runtime phase of component is a deployed component and the output is a component instance, which is expected to provide its clients with provided services when its required services are satisfied. The composition of components with fitting provided and required services resulting in a system is done at system design time when components are in deployment time. This composition cannot be changed any more at runtime. Therefore it is important to carefully select suitable components at system design time. As components are in deployment phase at system design time, it is essential for components to have a neat deployment contract to enable system designer to reasonably select components for their system. As shown above todays component definitions do not mention deployment contracts, which we consider essential.

Component update phase At this phase the component is updated with a new version of it. This phase is optional, which is illustrated in Figure 1 by a dotted line. If a system is not updated, its components are not updated either. The input for this phase is the old deployed component with a new component to be deployed. The output of the phase is the new deployed component. At this phase the new component has to be deployed to the system. So, the steps necessary to deploy a component have to be repeated. Those can include reregistering a new version of component with a naming service or providing a new component descriptor for the component container.

Component removal phase At this phase the component gets removed from the system. The steps that were necessary to deploy the component have to be reversed. Thus the input for this phase is a deployed component and the output is the vanishing of the component from the system.

Now that, we have defined all the basic concepts for our research and go on to elaborate on component metadata. Metadata in general is data describing data. The concept of metadata has been used throughout many areas in Computer Science. Database Systems rely on metadata to efficiently manage data stored in database management systems. Web services expose metadata to be discovered by web clients. Metadata is used in hardware systems to specify behaviour of their system parts. In the component-based software development, however, the use of metadata to specify behaviour of software components has not proliferated yet. In general, todays component models use interfaces to show the client how to use a component, which is considered a black box. The interface exposes syntactical contract of the component, i.e. method and event signatures offered by the component, but does not reveal any behavioural aspects inside the black box. This makes it difficult to reason about component's behaviour before integrating it into a system and impossible to differentiate between components possessing the same (syntactical) interface. Only few of the current component models use metadata to expose some component behaviour. And if at all then component metadata is intended to be used not by the component client but by the component's host, a container. Although current component models rarely use metadata for behavioural specification of components, some approaches have been developed that use metadata in related areas. The following survey unveils systems utilizing metadata for their components. Not all of them adopt one of the component definitions presented above. They are, however, included in the survey because of their contributions to exploiting component metadata.

3 Survey

In this chapter we go through all the systems we have been able to identify which use metadata for or about their components. We elaborate on each system and point out metadata usage.

3.1 ACME

ACME is an architecture description language defining connectors and components. A Component in ACME is an entity with ports, which connect to connectors. A Connector in ACME is an entity transferring control and data from ports of one component to the ones of another component. A system is created by connecting components to connectors and connectors to components resulting in a hierarchy of components communicating via connectors as shown in Figure 2. There is an implementation for ACME specifications called ArchJava [1, 2], which

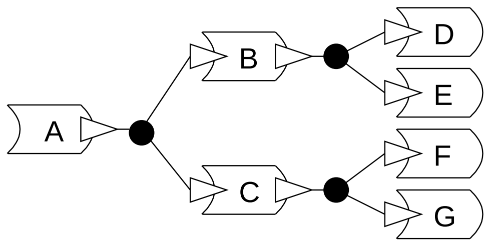


Figure 2: An ACME system

allows automatic transformation from ACME system specifications to Java source code that can be compiled and run.

An ACME extension presented in [23] allows adding metadata in form of annotations to ACME component specifications, which are used to perform static dependence analysis between components. Dependence analysis here is the study of how one element of the system can affect or be affected by other elements of the system. The annotations are in form of intra-component pathways connecting component input ports to output ports. Pathways capture the potential for an input to affect an output in some way.

Figure 3 shows an inter-component pathway description for a component X. The component has 6 ports: Start, 1, 2, 3, 4, and Exit. The path property indicates that an input on the Start port will cause either of the ports 3 or 4 to be triggered. Analogous, when an input on either port 1 or 2 is available, the port Exit will be triggered.

By using such component descriptions an automated dependence analysis is performed by the tool called Alladin. Chains of dependencies in architectural descriptions can be identified and it can be seen upfront what a replacement of a component by another one with different internal pathways would mean to the entire system. A list of ports with no source and those with no target can be identified as well.

```
Component X = {
    Port Start;
    Port 1;
    Port 2;
    Port 3;
    Port 4;
    Port Exit;
    Property paths = {
        [src="Start"; target="3";
                    relationship="causes"]
        [src="Start"; target="4";
                    relationship="causes"]
        [src="1"; target="Exit";
                    relationship="causes"]
        [src="2"; target="Exit";
                    relationship="causes"]
    };
};
```

Figure 3: ACME pathway description of component X

3.2 MetaH

MetaH [10, 11, 26] is an ADL and toolset developed to meet the requirements of flight control and avionics, including hard real time, safety, security, fault-tolerance and multiprocessing.

The essential concepts of MetaH are components and connections. Each component has a set of attributes, an interface and zero or more implementations. Connections link components together to form an architecture. MetaH specifications can also refer to a series of components called a path.

MetaH specifies how software modules are composed together with hardware objects to form a complete system architecture. MetaH specifications allow composing software objects such as subprograms, packages and processes and hardware objects such as memories and processors. MetaH allows system architects to integrate the source modules for all the various functional subsystems to form the final real-time, fault-tolerant, securely partitioned multi-processor system. Thus a MetaH system comprises both hardware and software parts and is a ready load image for a processor.

A concrete software component, or code module, in MetaH is any separately compiled unit in a programming language, e.g., a function or package. One or more software components are contained in a source file. Concrete software components can be grouped together to form more abstract components such as processes, modes and macros. A process groups subprograms together into a thread of control. The concrete hardware components in MetaH reflect the typical features of real-time embedded systems: processors, channels, memories, devices, etc. Hardware components can be grouped together into systems. Hardware and software components are grouped together at the highest level of MetaH abstraction: the application.

All components and their interface elements have attributes that specify values used in analysis and code generation. Some attributes of a component are inherited by all of its implementation parts.

Three forms of analysis have been defined for MetaH specifications: schedulability, reliability and safety/security.

Schedulability Analysis The schedulability analysis tool allows the system architect to determine whether an application can be feasibly scheduled (all processes can be dispatched at their specified periods or event arrival rates and complete by their specified deadlines). Schedulability analysis is based on the compute path and source execution time attributes given in the MetaH specification. This means that the schedulability analysis results are only as good as those attribute values.

Reliability Modelling The reliability modelling and analysis tools allow the system architect to determine the probability of failure of a fault-tolerant system that is subject to randomly arriving fault events. The exact set of faults, errors, and the response of various objects to faults can be specified by the system architect. The reliability analysis is based on 2 sources in a MetaH specification: error models and reliability-specific attributes. Error models define kinds of faults, errors and error behaviours. Attributes of objects can be set to make specific choices for specific objects, including fault event rates, propagation rates, error paths, and error-masking protocols (e.g., voting).

Safety/Security Modelling The safety/security modelling and analysis tool enables the system architect to determine if a specification is consistent with a stated set of safety/security attributes. Safety/security analysis insures that specified patterns of connections, data sharing, software/hardware binding and scheduling do not allow objects to interact in ways that violate a set of safety or security classifications for the individual objects. Objects in a specification can be assigned a safety level that ranges from A0 (highest safety level) to Z9 (lowest level of safety). A specification is unsafe if one object can affect in any way the proper operation of a second object when the second object has a higher security level than the first. A successfully completed safety analysis ensures that there are no mechanisms by which a defect in a lower-safety-level object could possibly affect the proper operation of a higher safety-level object. Safety levels can be assigned to hardware as well as software objects. A hardware object is considered to affect all software objects hosted on it for the purposes of safety/security analysis.

The procedure used to generate an executable image from a MetaH specification is as follows: The MetaH compiler transforms MetaH specifications into source code and compiler and linker directives. The application build tool (MakeH) uses information produced by the MetaH compiler to compile, link and, load images for the designated target hardware system (one for each processor). Example hardware specification in MetaH:

```
type package STANDARD is
```

```
    INTEGER: type;
    BOOLEAN: type;
    FLOAT: type;
end STANDARD;
type package implementation STANDARD.I80960MC is
attributes
    selfSourceName := STANDARD;
    INTEGERSourceDataSize := 4 B;
    BOOLEANSourceDataSize := 4 B;
    FLOATSourceDataSize := 4 B;
    INTEGERStatus := NONE;
    BOOLEANStatus := NONE;
    FLOATStatus := NONE;
end STANDARD.I80960MC;
```

Example software specification in MetaH:

```
with type package DOMAIN_TYPES;
process P1 is
    FROM_P2 : in port DOMAIN_TYPES.INTEGER_TYPE;
    TO_P2 : out port DOMAIN_TYPES.INTEGER_TYPE;
end P1;

periodic process implementation P1.SIMPLE is
attributes
    selfSourceTime := 100 us;
    selfPeriod := 1 sec;
    selfSourceFile := "p1.a";
end P1.SIMPLE;
```

Software components in MetaH are: port, event, package, monitor, package, subprogram, package, process. Attributes for software objects or components in MetaH include:

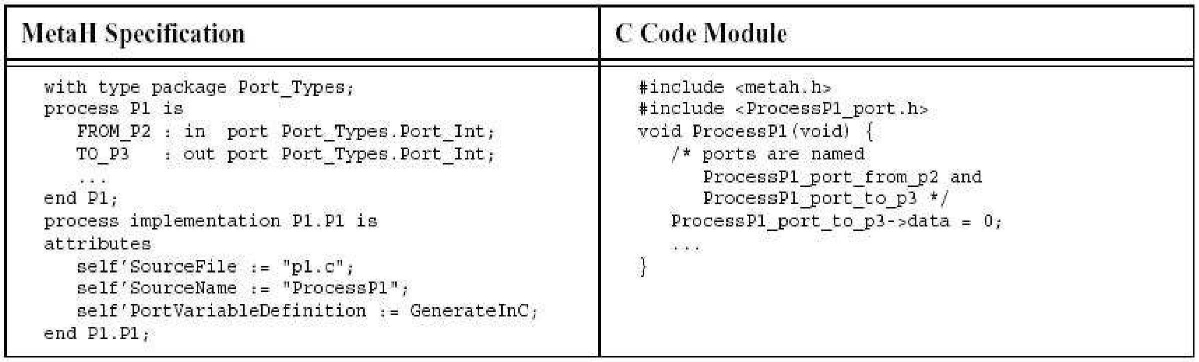
For Process: AllowedBinding, Criticality, Heapsize, Period, SourceFile, SourceName, SourceTime, StackSize, BuildOptions, SafetyLevel, ErrorPaths etc.

For Subprogram: AllowedBinding, Heapsize, SourceFile, SourceTime, StackSize, SafetyLevel, ErrorPaths, FaultEventRate etc.

For Event: SourceName, SafetyLevel, EventRate

For Application: SlowMotion, ErrorModel, ErrorPaths, FaultEventRate, Operable

Attributes are applied at design time of specification and used for code generation from the specification. So, the generated code depends on the attribute values. Figure 4 shows an example of a mapping of a MetaH specification to a C program. MetaH supports mappings to C and Ada.



MetaH Specification	C Code Module
see below	see below

MetaH Specification

```
with type package Port_Types;
process P1 is
   FROM_P2 : in  port Port_Types.Port_Int;
   TO_P3   : out port Port_Types.Port_Int;
   ...
end P1;
process implementation P1.P1 is
attributes
   self'SourceFile := "p1.c";
   self'SourceName := "ProcessP1";
   self'PortVariableDefinition := GenerateInC;
end P1.P1;
```

C Code Module

```
#include <metah.h>
#include <ProcessP1_port.h>
void ProcessP1(void) {
   /* ports are named
      ProcessP1_port_from_p2 and
      ProcessP1_port_to_p3 */
   ProcessP1_port_to_p3->data = 0;
   ...
}
```

Figure 4: A mapping from MetaH to C

3.3 Component metadata for software engineering tasks

A framework for defining metadata for components is outlined in [20]. It is argued that component developer implements a component that could be used in several, possibly unpredictable, contexts. Therefore, enough information has to be provided to make the component usable as widely as possible. In particular, the following information could be either needed or required by a generic user of a component:

Information to evaluate the component: for example, information on static and dynamic metrics computed on the components, such as cyclomatic complexity and coverage level achieved during testing

Information to deploy the component: for example, additional information on the interface of the component, such as preconditions, postconditions, and invariants

Information to test and debug the component: for example, a finite state machine representation of the component, regression test suites together with coverage data, and information about dependences between inputs and outputs

Information to analyze the component: for example, summary dataflow information, control flow graph representations of part of the component, and control-dependence information

Information on how to customize or extend the component: for example, a list of the properties of the component, a set of constraints on their values, and the methods to be used to modify them

Within the framework the idea behind MIME (Multi-purpose Internet Mail Extensions) types used for E-Mail attachments is drawn on for specifying metadata. A metadata type is defined as a tag composed of two parts: a type and a subtype, separated by a slash. Just like the MIME type "application/zip" tells a browser the type of the file downloaded in an unambiguous way, so the metadata type analysis/data-dependence could tell a component user (or a tool) the kind of metadata retrieved (and how to handle them). The actual information within the metadata can then be represented in any specific way.

So, the framework does not specify exact format of metadata but only provides a way of retrieving metadata types from components like e.g. test coverage.

Each component is provided with two additional methods: one to query about the kinds of metadata available, and the other to retrieve a specific kind of metadata. An example of those methods is as follows:

```
String[] component-name.getMetadataTags()
Metadata component-name.getMetadata(String tag, String[] params);
```

Using these methods the invariant for a component could be retrieved by executing

```
component-name.getMetadata("selfcheck/contract", params)
```

where params is an array of strings containing only the string "invariant", and post-condition for a method can be obtained by executing

```
component-name.getMetadata("selfcheck/contract", params)
```

where params is an array of strings containing two strings post and the method name, whose post-conditions should be obtained.

In [21] the framework outlined above is applied to store metadata about component's test coverage inside a component. The test coverage contains information about component's version, coverage measurement facilities, and changes between versions of components. Using this information, it is possible to derive test cases needed to be performed for regression testing of components.

3.4 CR-RIO

CR-RIO Framework [19] presents an approach to describe, deploy and manage component-based applications having dynamic functional and non-functional requirements. The approach is centred on architectural descriptions and associated high-level contracts. The latter allow the non-functional requirements to be described separately at design time, and during the runtime are used to guide architecture customizations required to enforce these requirements.

A contract in CR-RIO regulates non-functional aspects and can describe, at design time, the use of shared resources the application will make and acceptable variations regarding the availability of these resources. The contract will be imposed at run-time by an infrastructure for contracts enforcement.

A contract can have several contract categories. Contracts in CR-RIO are specified using Quality of Services Markup Language (QML). An example of a contract category in QML is as follows:

```
01 QoScategory Processing {
02  cpuUse: decreasing numeric %;
03  cpuSlice: increasing numeric %;
04  priority: increasing numeric;
05  memAvailable: increasing numeric Mbytes;
06  memReq: increasing numeric Mbytes;
07 }
```

The *Processing* category (lines 1-7) is one of quality of services of application. It represents processor and memory resources where the *cpuUse* property is the used percentage of the total CPU time (low values are preferred - *decreasing*), the *cpuSlice* property represents the time slice to be reserved / available to a given process (high values are preferred - *increasing*), priority represents a priority for its utilization, *memAvaliable* and *memReq* represent, respectively the available memory in the node and the memory (to be) requested for a process.

Other categories describe Data Transport (e.g. bandwidth, delay) and Data Replication:

```
QoScategory Replication {
 numberOfReplicas: increasing numeric;
 maxReplicas: increasing;
 replicaMaint: enum(add, remove, maintain};
 groupComm: enum (p2p, multicast, broadcast};
 distribPolicy: enum (bestMem, bestCpu, bestTransp, optim};
}
```

CR-RIO supporting middleware for contract enforcement is comprised of a Global Contract Manager (GCM) and Local Contract Managers (LCMs), Contractors and QoS Agents. The middleware uses QoS contracts, which are available as meta-level information, to instantiate an application and to manage its associated contract. The GCM represents the main authority; it can fully interpret and manage contract descriptions and knows their service negotiation state machine. When a negotiation is initiated the GCM identifies which service will be negotiated first and sends the related configuration descriptions, to each participating node, and the associated QoS problems to the LCM. Each LCM is responsible for interpreting the local configuration and activating a Contractor to perform actions such as resource reservation and method requests monitoring. If the GCM receives a positive confirmation from all LCM involved, the service being negotiated can be attended and the application can be instantiated with the required quality. If not, a new negotiation is attempted in order to deploy the next possible service. If all services in the negotiation clause are tried with no success, an out-of-service state is reached and a contract violation message is issued to the application level. The GCM can also initiate a new negotiation when it receives a notification informing that a preferred service became available again.

In the following we show a QoS contract for a replication configuration:

```
contract {
 service {
  instantiate server with profile ProcMem, Preplic;
  link client to server with profile Pcom;
 } repProc;
 negotiation {repProc -> out-of-service;};
}repServer;
profile {
  Processing.cpuSlice >= 0.25;
  Processing.memReq >= 200;
} ProcMem;
profile {
  Replication.numOfReplicas = 5;
  Replication.distribPolicy = optim;
} Preplic;
profile {
  Transport.delay < 5;
30   Replication.groupComm = multicast;
31 } Pcom;
```

According to the *repProc* contract each replica will only be instantiated if the *ProcMem* and *Preplic* profiles properties are satisfied. The number of replicas and the distribution policy described in the *Preplic* profile (lines 24-27) are controlled by the GCM. A number of five replicas were selected (line 25) and the distribution policy will try to optimize resources (line 26).

Contracts in CR-RIO are applied not to components but to systems parts, which are represented as nodes of a distributed application.

3.5 Jamus

The Jamus [22] framework enables the systems consisting of software components to dynamically contractualise their resource access conditions with their deployment environment. The system can draw on Raje (Resource-Aware Java Environment) which is a Java runtime environment that provides facilities to handle resources using objects.

Within contracts the behavioural dependencies binding components and their deployment environment regarding resource access conditions is defined and captured. Contracts can be used by components to provide their deployment environment with indication about the context in which they want to run. Contracts can also be used by deployment environments to inform components about the resource access conditions assigned to them (i.e. information about their execution context).

The resources in Jamus are classified in *observable*, *listenable*, *lockable*, *shareable*, *reservable* or *limitable*. According to this resource taxonomy objects are defined with specific operations to access the resources. Figure 5 illustrates objects for resources like CPU, Memory, Socket,

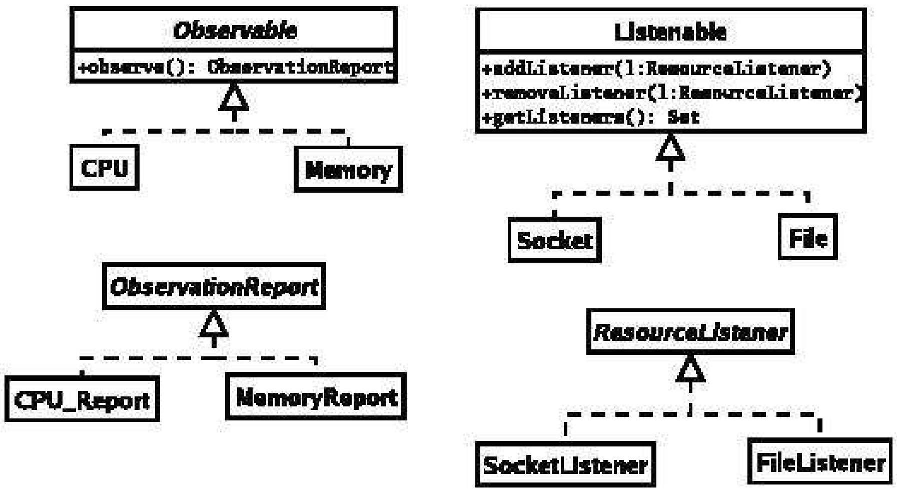


Figure 5: Object-based modelling for observable and listenable resources

File, CPU Report, Memory Report, Socket Listener as well as File Listener. Components can make use of these objects to access a resource.

Contracts can be used by components to provide their environment with indications about the context in which they want to run (i.e. about the resource access conditions they require). Contracts are defined as a set of so-called resource utilisation profiles. A resource utilisation profile basically aggregates four objects, which implement the *ResourcePattern*, *ResourcePermission*, *ResourceQuota* and *ResourceAvailabilityConstraints* interfaces respectively.

In the following we show an example of a resource utilization profile. Some requirements are imposed: a requirement for connections to the specified Web server: 15 MB received, 1 MB sent and requirement concerning access to directory /opt/music: 20 MB read, 20 MB written.

```
int MB = 1024*1024;
int KB = 1024;

ResourceUtilisationProfile R1,R2;

// Selective requirement for connections to the specified Web server:
// 15 MB received, 1 MB sent.
R1 = new ResourceUtilisationProfile(
       new SocketPattern("http://www.music.com"}),
       new SocketPermission(SocketPermission.ALL),
       new SocketQuota(15*MB, 1*MB), new BestE.ort());

// Selective requirement concerning access to directory /opt/music:
//20 MByte read, 20 MB written.
R2 = new ResourceUtilisationProfile(new FilePattern("/opt/music"),
       new FilePermission(FilePermission.WRITE ONLY),
       new FileQuota(20*MB, 20*MB),
       new ResourceReservation());

ResourceOrientedContract contract1 = new ResourceOrientedContract ({R1,R2});
```

Once an object includes such contracts in its code, they are negotiated at runtime before the object is allowed to access a specific contractually-requested resource.

3.6 Composition Support with lightweight metadata-based extensions of Component Models

The authors of [16] use metadata attributes to assign additional information to components in .NET component model and validate component compositions.

The metadata attributes are defined to

- mark component's required interfaces
- provide information on constraint checks for method invocations
- check if all participants in component collaboration satisfy a certain protocol

In the following we show the attribute 'Required' applied to a property 'RequiredProperty'. All properties bearing the 'Required' attribute must be set before a component instance can be used.

```
public class Test : Control
{
 private IMyRequiredInterface required;
 [Required(typeof(IMyInterfaceA))]
 [Required(typeof(IMyInterfaceB))]
 public IMyInterfaceA RequiredProperty
 {
  get { return this.required; }
  set { return this.required = value; }
 }

 // paint method checks all required properties
 public override void OnPaint(PaintEventArgs e)
 {
  if (!RequiredHelper.CheckAllRequiredProps(this))
  { // paint error message }
  else
  { // normal drawing code }
 }
}
```

A component can be instantiated by a designer to allow for graphical component composition. The designer will call the OnPaint method to visualize component. This method contains validation code for checking the attributes. If not all the properties marked with the 'Required' attribute are set, the component will not be displayed properly. The same happens at runtime of the system. Thus, the code for checking the attributes is inside component's code and the attributes themselves are for validating the proper usage of the component.

Constraint checks are specified using invariants with pre- and post conditions expressed by 'InvariantAttribute', 'PreAttribute' and 'PostAttribute' attributes. The attributes contain Object Constraint Language (OCL) expressions, which can be checked by the corresponding checker.

The following example illustrates an example of an OCL expression inside pre- and post-condition attributes. Checks can be performed inside components code, i.e. at runtime, using a provided OCLChecker.

```
public class Account
{
 int balance;

 [Pre("amount >= 0 and self.balance >= amount")]
 [Post("self.balance = self.balance@pre - amount")]
 public void Withdraw (int amount)
 {
  this.balance -= amount;
 }
}

// check code
OCLCheck.PreconditionCheck(this, "Withdraw", increment);
```

Finally, the attributes for specifying collaboration protocols are presented in the work. Protocols define a predefined order for access of methods and properties, i.e. state machines for ordering method invocations. Protocols are assigned to interfaces. A protocol attribute takes a protocol name, an array of state names of the state machine, and the initial state. Other interfaces taking part in the protocol are marked with Collaborator attributes that are initialized with the protocol name and the type of the participating interface. For each method Transition attributes are used to declare allowed state transitions associated with the invocations. Each transition attribute is initialized with the name of the source state and the target state.

In the following we show an example of two interfaces that share the access to a file. The protocol check shown at the end ensures that a file must be opened before it can be accessed for reading. Calls to the StateMachine must be inserted inside component at the beginning of methods taking part in the protocol to ensure that the StateMachine is in a valid state before proceeding with method execution. Each protocol requires a state machine to be provided by the protocol developer.

```
[Protocol("Interaction", new string[]{"Closed", "Open"}, Initial="Closed")]
[Collaborator(typeof(I2))]
public interface IProvider
{
 [Transition("Closed", "Open")]
 void Open();

 [Transition("Open", "Closed")]
 void Close();
}

public interface I2
[Protocol("Interaction", new string[]{"Closed", "Open"}, Initial="Closed")]
[Collaborator(typeof(I1))]
public interface IReader
{
 [Transition("Open", "Open")]
 object Read();
}

// check code
StateMachine.Check(this, "Read");
```

To sum it up, this work exploits design by contract paradigm in that the attributes attached to the component are applied at design time of component and checked at runtime. The component's code will probably contain lots of checking code for checking if all the required interfaces of component are set, whether pre-, postconditions and invariants hold true and whether the state machine for a protocol is in a valid state for performing an operation.

3.7 EJB

Enterprise Java Beans (EJB) [13, 8, 5, 14, 25] is a component model for building (collaborating) enterprise java beans. The enterprise beans in EJB component model are running in an EJB container. Each bean has to be deployed into the EJB container using a deployment descriptor. The EJB container manages enterprise beans throughout their lifecycle and can perform some actions on behalf of the beans hosted. In order for the container to be able to manage the beans properly, every bean has to provide the container with a deployment descriptor containing information as to what and how to manage. In most cases, specifying a resource in a deployment descriptor will enable an enterprise bean to reference the resource from within its code using a logical name. The mapping from the logical name of the resource to the resource itself is managed by the container. A deployment descriptor is written in XML and is human-readable and changeable. There are two basic kinds of information in the deployment descriptor:

Enterprise bean's structural information describing an enterprise bean and its external dependencies. The structural information is mandatory and cannot be changed because doing so could break the enterprise bean's function

Application assembly information describes how the enterprise bean (or beans) is composed into a larger application deployment unit. Assembly level information is optional and can be changed without breaking the enterprise bean's function, although doing so may alter the behaviour of an assembled structure of the bean

A deployment descriptor contains the following metadata about enterprise bean(s):

Re-entrancy indication The Bean Provider must specify whether an entity bean is re-entrant or not. The container will forbid entering a bean simultaneously if it is marked non re-entrant in the deployment descriptor.

Session beans state management type If the enterprise bean is a session bean, the Bean Provider must use the session-type element to declare whether the session bean is stateful or stateless:

```
<session-type>Stateful</session-type>
<session-type>Stateless</session-type>
```

This has implications on persistence management of the bean.

Entity beans persistence management If the enterprise bean is an entity bean, the Bean Provider must use the persistence-type element to declare whether persistence management is performed by the enterprise bean or by the container.

```
<persistence-type>Bean</persistence-type>
<persistence-type>Container</persistence-type>
```

Entity beans primary key class If the enterprise bean is an entity bean, the Bean Provider specifies the fully-qualified name of the entity beans primary key class in the prim-key-class element. The Bean Provider must specify the primary key class for an entity with bean-managed persistence.

```
<primkey-field>EmployeeId</primkey-field>
```

Container-managed fields If the enterprise bean is an entity bean with container-managed persistence, the Bean Provider must specify the container-managed fields using the cmp-field elements. An entity bean with container-managed persistence relies on the container to perform persistent data access on behalf of the entity bean instances. The container transfers data between an entity bean instance and the underlying resource manager. The container also implements the creation, removal, and lookup of the entity object in the underlying database.

Container-managed relationships If the enterprise bean is an entity bean with container-managed persistence and cmp-version 2.x, the Bean Provider must specify the container-managed relationships of the entity bean using the relationships element. The container maintains the relationships among entity beans. It is the responsibility of the container to maintain the referential integrity of the container-managed relationships in accordance with the semantics of the relationship type as specified in the deployment descriptor.

Environment entries The Bean Provider must declare all the enterprise bean's environment entries. The container takes care of setting the environment variable so that the bean can make use of it. An Example of declaring an environment variable is shown in the following:

```
<env-entry>
    <description>Some description.</description>
    <env-entry-name>name3</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
```

Resource manager connection factory references The Bean Provider must declare all the enterprise bean's resource manager connection factory references. The resource manager can manage data base connections for the beans. All the managed database connections must be specified in the deployment descriptor. An example of a data base connection in an EJB deployment descriptor is shown in the following XML fragment:

```
...
<enterprise-beans>
   <session>
       ...
       <ejb-name>EmployeeService</ejb-name>
       <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
       ...
       <resource-ref>
           <description>
               A data source for the database in which
               the EmployeeService enterprise bean will
               record a log of all transactions.
           </description>
           <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
```

```
            <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>
        ...
    </session>
</enterprise-beans>
...
```

EJB references The Bean Provider must declare all the enterprise bean's references to the remote homes of other enterprise beans. An example of an enterprise bean specifying some references to dependent beans is depicted in the following XML. Assembly information of enterprise beans comprising an application is specified using a bean's references to other beans.

```
...
<enterprise-beans>
   <session>
        ...
        <ejb-name>EmployeeService</ejb-name>
        <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
        ...
        <ejb-ref>
            <description>
                This is a reference to the entity bean that
                encapsulates access to employee records.
            </description>
            <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.wombat.empl.EmployeeRecordHome</home>
            <remote>com.wombat.empl.EmployeeRecord</remote>
        </ejb-ref>

        <ejb-ref>
            <ejb-ref-name>ejb/Payroll</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.aardvark.payroll.PayrollHome</home>
            <remote>com.aardvark.payroll.Payroll</remote>
        </ejb-ref>

        <ejb-ref>
            <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>com.wombat.empl.PensionPlanHome</home>
            <remote>com.wombat.empl.PensionPlan</remote>
        </ejb-ref>
        ...
   </session>
   ...
```

```
</enterprise-beans>
...
```

Web service references The Bean Provider must declare all the enterprise bean's references to web service interfaces. The EJB container establishes a connection to the web servers specified in the deployment descriptor. Each bean has to specify its web service references. I.e. several beans may need to reference the same web service, which will not cause name conflicts. A web service reference to a web service with the name 'service/StockQuoteService' can be specified as follows in the EJB deployment descriptor:

```
<session>
   ...
   <ejb-name>InvestmentBean</ejb-name>
   <ejb-class>com.wombat.empl.InvestmentBean</ejb-class>
   ...
   <service-ref>
       <description>
           This is a reference to the stock quote
           service used to estimate portfolio value.
       </description>
       <service-ref-name>service/StockQuoteService</service- ref-name>
       <service-interface>com.example.StockQuoteService</service-interface>
   </service-ref>
   ...
</session>
```

Message destination references The Bean Provider must declare all the enterprise bean's references to message destinations. A message destination reference refers to a resource capable of receiving messages, like e.g. Java Message Service (JMS). The EJB container establishes a connection to all method destination references found in an enterprise bean's deployment descriptor. Specification of JMS as a message sink in an EJB deployment descriptor is done as follows:

```
...
<message-destination-ref>
 <description>
   This is a reference to a JMS queue used in processing Stock info
 </description>
 <message-destination-ref-name>jms/StockInfo</message-destination-ref-name>
 <message-destination-type>javax.jms.Queue</message-destination-type>
 <message-destination-usage>Produces</message-destination-usage>
</message-destination-ref>
...
```

A message destination reference is scoped to the enterprise bean whose declaration contains the <message-destination-ref> element. This means that the message destination reference is not accessible to other enterprise beans at runtime, and that other enterprise beans may define <message-destination-ref> elements with the same <message-destination-ref-name> without causing a name conflict.

Security The Bean Provider must declare security requirements on the caller of its enterprise bean. This is done by various corresponding entries in the deployment descriptor. The security infrastructure ensures at runtime that the caller can fulfil the security requirements, otherwise the call is rejected.

Besides deployment descriptor there is also assembly descriptor in EJB for defining application-assembly information. The application assembly information consists of the following parts: the definition of security roles, the definition of method permissions, the definition of transaction attributes for enterprise beans with container managed transaction demarcation and a list of methods to be excluded from being invoked. Providing an assembly-descriptor in the deployment descriptor is optional. The assembly descriptor itself does not introduce new metadata about components. It rather uses the set of metadata from the deployment descriptor.

3.7.1 Summary

Having metadata about component specified in XML has a drawback that is can be tampered with. XML is a human-readable format and can easily be changed maliciously, which is fatal for correct system execution.

Furthermore, the metadata about enterprise beans is meant for the container and not for compositional reasoning. Application assembly information and metadata are completely separate in the deployment descriptor. EJB does not consider correlations between metadata of assembled beans.

Additionally, an EJB container has a limited set of manageable features, which can be specified by an enterprise bean in its deployment descriptor. Thus a bean specifies metadata about itself in the deployment descriptor for the container only and not in full detail. I.e. an enterprise bean can have far more dependencies to the environment or other important properties than specified in its deployment descriptor. In other words, information in the deployment descriptor about bean's behaviour is not comprehensive and not sufficient for compositional reasoning.

3.8 CCM

CORBA Component Model (CCM) [18, 28, 17] is a component model which allows creating CORBA components. Components in CORBA component model (CCM) are deployed into, managed by and run in a container. A component must have a component descriptor written in XML in order to be loaded into and by the container. As the XML is similar to the one used in EJB, presented in Section 3.7, we abandon it here and only present the kinds of metadata available in CCM.

The CORBA Component descriptor describes a component and contains the following metadata about it:

Component kind A component kind can be session, service, process, or entity. The kind of component determines the kind of container the component must reside in.

Transaction management There are several ways of handling transactions by a CCM container. A component can indicate not to engage in transactions, to manage them on its own or to get them managed by the container.

Security component security settings of component specify CORBA security rights for the component. The container enforces security policy by not allowing invocations without appropriate rights to take place.

Thread safety indication A component can indicate whether it is thread-safe or not. This serves as an indication for the container about how to dispatch operations on the component instance. If threading is set to multithread, then the component is ready to accept multiple threads of control within a single instance. The component takes responsibility for protecting its internal state. If threading is set to serialize, then the container will serialize all calls to a single instance. Although the component will not need to protect instance state in this case, the container may employ other threads to invoke other instances of the component type. Thus the component must protect any static or class data.

Container managed persistence A component can indicate whether it wants to use CORBA persistent state service or other user-defined persistence mechanism. The container manages the relationship between component state to be persisted and the associated persistent state service.

Interoperability with other component types a component can specify if it interoperates with other component types like e.g. EJB components. A component can act as a view for another component type or have a view of that type.

Besides component descriptors describing one component there are component assembly descriptors in CCM, which describe an assembly of components. The component assembly descriptor describes which components make up the assembly, how those components are partitioned, and how they are connected to each other. A component assembly descriptor is the recipe for deploying a set of interconnected components. It is alike the component descriptor expressed in XML. Assembly descriptor contains information about events exchanged among components and the way they are distributed by components in the assembly. The container takes care of registering the components exchanging events with CORBA event infrastructure. Furthermore, a component specifies connections to interfaces it requires and states interfaces it provides for other components. The container takes charge of ensuring that all required and provided interfaces stated in the assembly descriptor are existent. The assembly descriptor itself does not contain additional metadata than stated in the component descriptor of each component from the assembly.

3.9 .NET Component Model

.NET Component Model is a part of .NET Framework [4, 15]. It allows creating .NET components, which can be deployed into .NET applications using a graphical application designer that is integrated into the .NET IDE - Microsoft Visual Studio .NET [12, 3, 27].

The .NET Component Model lets their components draw on predefined attributes for different purposes. The attributes are metadata about component behaviour. The attributes are evaluated by the underlying .NET framework and Common Language Runtime (CLR). The attributes are parameterizable entities, which can be assigned to an assembly, which is a unit of deployment in .NET, a class, a property, a method, a method parameter or a method return value. Overall, there are 195 attributes available in the categories illustrated in the Table 1. All of the attributes are applied at component design time. The time of attribute processing differs among attribute categories depicted in Table 1. Attributes in categories 1, 3, 5, 7 and partly 10 are processed at component compilation time. Attributes in categories 4, 6, 8, 9, 11 and partly 10 are processed at component runtime. The only attribute category that is processed at component deployment time is 2.

Nr	Attribute category	Purpose
1	System.CLSCompliantAttribute	Ensures all the types in an assembly be CLS compliant
2	System.ComponentModel category	Provide the graphical component designer with information regarding component treatment at application design-time
3	System.Diagnostics category	Provide the compiler with information on whether to generate debug symbols during compilation
4	System.EnterpriseServices category	Provide the .NET CLR with information on how to treat classes communicating with COM+ components
5	System.Management.Instrumentation category	Advise the compiler on whether to instrument certain code details
6	System.Reflection category	Provide general information about an assembly, which can be reflected
7	System.Runtime.CompilerServices category	Provide the compiler with information on treating some code blocks in a special way
8	System.Runtime.InteropServices category	Provide the .NET CLR with information on how to treat classes which communicate with COM objects and vice versa
9	System.Security category	Provide the CLR with security information assigned to code fragments
10	System.Web.Services category	Attributes supporting web services construction
11	System.Xml.Serialization category	Attributes supporting XML serialization of classes

Table 1: Attributes available in .NET

4 Analysis

In the following we provide an analysis of systems using metadata presented above. We choose the following criteria for classifying the systems:

Metadata inside or outside component This criterion indicates whether a component has metadata inside it or not. Some of the systems in the taxonomy apply metadata to component specifications or put them into component descriptions. Other systems put metadata inside components themselves.

Metadata used for some component behaviour analysis This criterion indicates whether metadata is used for an analysis (by a tool) in the system. If metadata is used for only communicating the container what to do with the component at runtime like in EJB or CCM it is not considered to be analysis. On the contrary if metadata is used for deriving some information about component behaviour which is subsequently analysed to derive some more information about system behaviour, it is considered to be analysis.

Component lifecycle phase of metadata processing This criterion says whether metadata is processed at component design, deployment or runtime. In this criterion unlike in the previous one any evaluation of metadata is considered. It is used to identify the component lifecycle phase the analysis takes place in.

Metadata specification method This criterion describes the method used to specify component metadata. Some systems use XML to specify metadata, others use special types or attributes.

Metadata used for some component composition analysis This criterion indicates whether metadata of individual components is used to predict some behaviour of some composition of those components. In other words, this section points out whether some kind of predictable assembly of components by analysing component metadata is supported by the system.

A classified survey of systems utilizing component metadata to accomplish their tasks is presented in Table 2. The systems are classified according to the criteria outlined above.

Row 1 of Table 2 shows that today's ADLs like ACME and MetsH put metadata outside components. In fact, they are found in component's specifications. So, do EJB, CCM component models. Metadata about components is found component deployment descriptors that are used to deploy a component to a container. This, however, has a drawback that metadata can be tampered with on component deployment resulting in wrong component management by the container. .NET component model puts metadata inside components and thus avoids the possibility of easy corruption of metadata.

Row 2 of Table 2 shows that ADLs use metadata for component behaviour analysis. Especially MetaH has an extensive set of metadata for specifying component behaviour. Today's component models EJB, CCM and .NET, however, do not use metadata to analyse component behaviour.

Row 3 of Table 2 shows that only ADLs process metadata at component design time. In fact, current approaches to software architecture are mostly concerned with designing of systems. As a corollary when designing systems, ADLs specify components (boxes) and connectors (lines between components). In other words, component design and system design is one process in ADLs.

Row 4 of Table 2 shows that only .NET component model processes component metadata at component deployment time, and only in a limited way as outlined in Section 3.9. Surprisingly, other component models, EJB and CCM, do not process metadata at component deployment time despite component deployment descriptors. A Component Deployment Descriptor contains information about the component itself, like its name, interfaces, class name, component type etc., as well as information as to how the component to be managed by the container, which is the actual metadata of the component. The metadata is processed by the container at component runtime but not deployment time. The term deployment descriptor can be justified as the descriptor is used on component deployment to the container before runtime. Its content with respect to metadata is processed at runtime, though.

Row 5 of Table 2 shows that in contrast to deployment time component metadata is processed at runtime by all systems except ADLs.

Row 6 of Table 2 denotes how metadata is specified. ADLs use their specifications to put metadata in. The work on metadata for software cngineering tasks from Section 3.3 suggests using MIME types for specifying component metadata. This is the approach taken for specifying types of E-Mail attachments. CR-RIO framework uses a special language, QML. Jamus specifies

Nr	*Property*	*ACME Ext.*	*MetaH ADL*	*Meta data forSW Engin.*	*CR – RIO*	*Ja mus*	*Comp. Model Ext.*	*EJB*	*CCM*	*.NET*
1	*Metadata in/outside component*	*Out side*	*Out side*	*In side*	*Out side*	*In side*	*In side*	*Out side*	*Out side*	*In side*
2	*Component behaviour analysis*	*Yes*	*Yes*	*Yes*	*Yes*	*No*	*Yes*	*No*	*No*	*No*
3	*Metadata processed@ component design*	*Yes*	*Yes*	*Not speci– fied*	*No*	*No*	*No*	*No*	*No*	*No*
4	*Metadata processed@ component deployment*	*No*	*No*	*Not speci– fied*	*No*	*No*	*No*	*No*	*No*	*Yes*
5	*Metadata processed@ component runtime*	*No*	*No*	*Not speci– fied*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
6	*Metadata specifi– cation method*	*Anno ta– tions in comp. spec.*	*Attrib utes in comp. spec.*	*MI ME types*	*QoS Mark up Lang. QML*	*Spe cial types avai lable*	*Attrib utes on comp.*	*XML comp. desc– rip– tors*	*XML comp. desc– rip– tors*	*Attrib utes on comp.*
7	*Metadata usedfor component composition analysis*	*Yes*	*Yes*	*Not speci– fied*	*No*	*No*	*No*	*No*	*No*	*No*

Table 2: A classified survey of systems employing component metadata

special types denoting component metadata. EJB and CCM component models use XML files corresponding to an XML schema [6]. .NET component model as well as the work on composition support with lightweight metadata-based extensions of component models from Section 3.6 use an interesting style to specify metadata. They draw on a facility offered by .NET and Java for applying attributes to types. Attributes are special types, which can be applies to other types. They are in fact like aspects of the type they are applies to. They are compiled into the binary on component compilation and can be retrieved from there even before component instantiation.

Row 7 of Table 2 shows that compositional reasoning of components is done only in ADLs. None of component models performs compositional reasoning of components, which is an open research area today.

5 Conclusion

In this report we have shown a number of approaches to utilizing component metadata to accomplish various tasks. We have disclosed investigated areas as well as those that need to be researched yet. The analysis shows clearly that in the context of component models[7, 9] much remains to be done to make exhaustive use of metadata.

Our research group is working on a new component model which would allow compositional reasoning of components at component deployment time. As summarized in Table 3 none of

Component composition analysis in component deployment phase	
Systems from the survey	No
Proposed Component Model	Yes

Table 3: Component composition analysis at component deployment time

the systems surveyd here can do this.

This report can be used as a starting point for research projects concerned with component metadata.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proc. 16th European Conference on Object-Oriented Programming*, pages 334–367. Springer-Verlag, 2002.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implimentation. In *Proc. ICSE 2002*, pages 187–197. IEEE, 2002.

[3] Microsoft Corporation. *Microsoft Visual Studio.NET.* Microsoft Press, 1st edition, 2001.

[4] Microsoft Corporation. Msdn .net framework class library version 2.0, 2005.

[5] L.G. DeMichiel, L.Ü. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans Specification Version 2.0*, 2001.

[6] L. Dykes, E. Tittel, and C. Valentine. *XML Schemas.* Sybex Inc, 2002.

[7] J. Estublier and J.M. Favre. Component models and technology. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-Based Software Systems*, pages 57–86. Artech House, 2002.

[8] Richard Monson Haefel. *Enterprise Java Beans.* O'Reilly, 4th edition, 2004.

[9] K.-K. Lau and Z. Wang. A survey of software component models. Pre-print CSPP-30, School of Computer Science, The University of Manchester, April 2005. `http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp30.pdf`.

[10] B. Lewis. Software portability gains realized with metah and ada95. In *Proceedings of the 11th international Workshop on Real-Time Ada Workshop*, 2002.

[11] B. Lewis, E. Colbert, and S. Vestal. Developing evolvable, embedded, time-critical systems with metah. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, 2000.

[12] Craig Skibo Marc Young, Brian Johnson. *Inside Microsoft Visual Studio .NET 2003*. Microsoft Press, 2nd edition, 2003.

[13] Sun Microsystems. Enterprise java beans specification, version 3.0, 2005.

[14] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 4th edition, 2004.

[15] Adam Nathan. *.NET and COM: The Complete Interoperability Guide*. Sams, 1st edition, 2002.

[16] Johann Oberleitner and Michael Fischer. Improving composition support with lightweight metadata-based extensions of component models. In *Software Composition*, 2005.

[17] ObjectWeb – Open Source Middleware. *OpenCCM User's Guide*. `http://openccm.objectweb.org/doc/0.8.1/user_guide.html`.

[18] Object Management Group (OMG). Corba components, specification, version 0.9.0, 2005.

[19] Alexandre Sztajnberg Orlando Loques. Customizing component-based architectures by contract. In *Component Deployment: Second International Working Conference*, volume 3083, pages 18–34. Lecture Notes in Computer Science, January 2004.

[20] Alessandro Orso, Mary Jean Harrold, and David S. Rosenblum. Component metadata for software engineering tasks. In *Revised Papers from the Second International Workshop on Engineering Distributed Objects*, pages 129–144, 2000.

[21] Alessandro Orso, Mary Jean Harrold, David S. Rosenblum, and Gregg Rothermel. Using component metadata to support the regression testing of component-based software. Technical Report GIT-CC-01-38, College of Computing, Georgia Institute of Technology, 2001.

[22] Nicolas Le Sommer. Towards a dynamic resource contractualisation for software components. In *Component Deployment: Second International Working Conference*, volume 3083, pages 129–143. Lecture Notes in Computer Science, January 2004.

[23] J. A. Stafford and A. L. Wolf. Annotating components to support component-based static analyses of software systems. In *Procoeedigns of the Grace Hopper Celeb. of Women in Computing*, 2001.

[24] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[25] T. Valesky and T.C. Valesky. *Enterprise JavaBeans: Developing Component-Based Distributed Applications.* Addison-Wesley, 1999.

[26] S. Vestal. Metah programmer's manual, version 1.09. Technical report, 1996.

[27] Microsoft Visual Studio Developer Center. `http://msdn.microsoft.com/vstudio/`.

[28] Ron Zahavi. *Enterprise Application Integration with CORBA Component and Web-Based Solutions.* John Wiley and Sons, 1st edition, 1999.