Analyzing and Understanding Architectural Characteristics of COM+ Components

Martin Pinzger, Johann Oberleitner, and Harald Gall
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
{pinzger, joe, gall}@infosys.tuwien.ac.at

Abstract

Understanding architectural characteristics of software components that constitute distributed systems is crucial for maintaining and evolving them. One component framework heavily used for developing component-based software systems is Microsoft's COM+. In this paper we particularly concentrate on the analysis of COM+ components and introduce an iterative and interactive approach that combines component inspection techniques with source code analysis to obtain a complete abstract model of each COM+ component. The model describes important architectural characteristics such as transactions, security, and persistency, as well as create and use dependencies between components, and maps these higher-level concepts down to their implementation in source files. Based on the model, engineers can browse the software system's COM+ components and navigate from the list of architectural characteristics to the corresponding source code statements. We also discuss the Island Hopper application with which our approach has been validated.

Keywords: reverse engineering, software components, COM+, component inspection

1. Introduction

Large applications are often realized by using a three-tiered architecture that separates the presentation logic, business logic and data access logic from each other. This separation improves scalability, fault-tolerance and evolvability of the whole application since single tiers can be replicated and hence bottlenecks can be avoided [3, 14].

Current component technologies such as COM+ [10] and Enterprise JavaBeans (EJB) [17, 24] provide additional techniques such as load-balancing, transaction handling and role-based access control. Hence, using a distributed component model has been a natural means to implement the non-visual parts of a three-tiered architecture. The client often accesses the presentation tier via a web browser. The presentation logic that renders this web content is based on web languages such as Microsoft's Active Server Pages (ASP), Common Gateway Interface (CGI) scripts or Java Server Pages (JSPs). Sometimes clients can use native applications that still use distributed component technology.

Vendors of component frameworks provide specific support for realizing three-tiered applications, in particular for the implementation of transactions, security, state, and persistence. These architectural concepts are common to current component models. However, although the architectures of three-tiered applications can be developed generically, a major drawback of vendor-specific solutions is that they are not portable. Once a framework has been selected, such as COM+, it is not easy to switch to another, potentially more advanced component framework.

Our overall goal is to realize a semi-automatic transformation of three-tiered applications between different frameworks such as for example from COM+ to EJB or .NET based applications. To perform this transformation an understanding of the architectural characteristics of the components comprising such applications is fundamental. In this context *architectural characteristics* describe the realization of architectural concepts such as transactions, security, persistency, and create and use dependencies between components. In this paper we concentrate on the analysis process for extracting architectural characteristics of COM+ components that are used in the business tier and the database tier of three-tiered applications.

A prerequisite for the understanding of software components is the availability of an abstract representation describing the architectural characteristics of each component.

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

Various tools in both the commercial and research community are available that address the abstraction problem by analyzing source code and run-time information. Some examples are Rigi [25], Dali [9], Software Bookshelf [5] as well as Imagix-4D [7] or SourceNavigator [23]. Although they are used to understand software systems they don't concentrate on specific application domains such as understanding the architectural characteristics of COM+ components, in which additional meta and configuration data has to be considered.

In this paper we address the application domain of three-tiered software systems built of COM+ components and introduce an iterative and interactive analysis approach that combines traditional source code analysis with the analysis of component specific meta and configuration data. The result is an abstract model that for each COM+ component describes important architectural characteristics including transactions, security, and persistency, as well as create and use dependencies between components, and maps these higher-level concepts down to their implementation in source files. Based on the model engineers can then browse the software system's COM+ components and navigate from the list of architectural characteristics to the corresponding source code statements.

The remainder of the paper is organized as follows. Section 2 provides some required background information about the COM/COM+ component model. The data model and its generation by analyzing source code, meta data, and configuration details are described in Section 3. Section 4 demonstrates the use of our analysis tools by a case study and mentions limitations of our approach. Related work is presented in Section 5 and Section 6 summarizes the paper and indicates future work.

2. COM/COM+ components

This section provides the necessary technical foundation of Microsoft's Component Object Model (COM) [2, 4, 16] and its recent enhancement COM+ [10] to facilitate the understanding of our approach.

COM is a component model [6] used heavily within Microsoft's operating systems. Like other component models the only way to access a component's functionality is through explicit interfaces exposed to the outside. These interfaces can be declared with Microsoft's Interface Definition Language (MIDL), a language that allows the definition of interfaces and component classes out of a set of primitive language constructs similar to that of C++. As usual interfaces consist only of functions, component classes consist of sets of incoming or outgoing interfaces. Latter interfaces are not implemented by COM components but used to issue events.

The MIDL compiler uses an IDL file to generate a COM type library and some C-/C++ header files for reuse of the component in clients. Microsoft VisualBasic does not create IDL files but automatically builds a type library file when compiling a COM component. The type library contains the same information as the IDL file but uses a binary representation and can be queried by predefined COM interfaces. Frequently, type libraries are provided together with the COM binary files (DLL or EXE).

COM components are always used as binary units that can be deployed on a system by putting it into one persistent location and registering the component in the Windows Registry. The registration information contains data where to find the DLL or EXE file, the type library, versioning information and other data. Due the binary representation there are only two ways to access a component's interfaces: either the clients use interfaces they are aware of at build time or they use the component's dynamic invocation service realized by the `IDispatch` interface. In both cases the interfaces of the component are the only way to access a component's functionality.

COM+ enhances COM with server-side facilities such as transaction monitoring or event filtering. These new mechanisms have been integrated in Windows 2000 and Windows XP. It is not restricted solely to the use of programmatic transactions but allows also the configuration of transactional behavior of COM+ components. Hence, it is possible to automatically create new transactions just by configuring components. In addition COM+ introduced a powerful role-based access-control scheme to make COM+ based applications secure. In addition to programmatic security it is possible to set a component's access rights at the component-level, at the interface-level, and at the method-level. The COM+ services implemented by Windows 2000 or Windows XP also increase scalability of components with an activation/passivation mechanism applied to component instances not used regularly.

Unlike for traditional COM these settings mentioned above do no longer rely just on the Windows Registry but Microsoft has introduced a set of COM+ administration components that allow querying and modification of declarative transaction and security settings.

COM+ has not become obsolete by Microsoft's .NET initiative since .NET does not introduce new server-side facilities that replace COM+ [26]. On the contrary the COM+ services work together with .NET. Frequent programming models for COM and COM+ are based on Microsoft VisualBasic or Microsoft's Visual C++ compiler using the Active Template Library (ATL) and Microsoft Foundation Classes (MFC).

In this work we restricted the source code analysis to VisualC++, VisualBasic, and the frameworks mentioned. Most analysis schemas can be transformed to other COM programming environments, too.

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

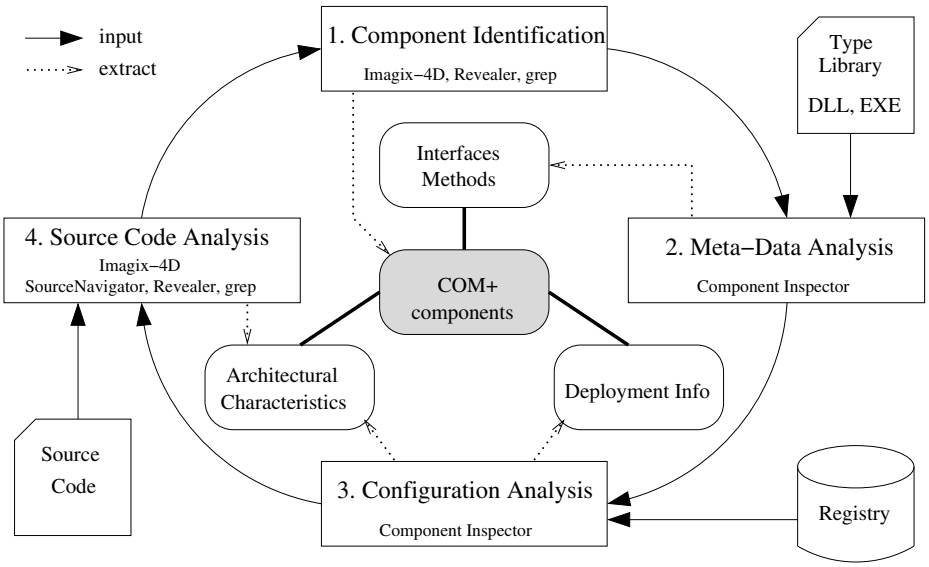

Figure 1. COM+ component analysis process.

3. Analyzing COM+ components

The goal of our analysis process is the generation of an abstract model describing important architectural characteristics of each COM+ component. A list of architectural characteristics we currently take into account is given in Table 1. To provide a complete picture of a COM+ component we have to consider different information sources including:

- *Source code* comprises definition (IDL) and source files that define and implement the interfaces of COM+ components. Source files also contain the statements that indicate certain architectural characteristics such as for example `SetComplete()` and `SetAbort()` for handling transactions.
- *Type libraries* correspond to interface definition files and provide detailed information about the interfaces implemented and provided by COM+ components. Instead of parsing IDL files we use the COM+ API to extract the type library information of each COM+ component. In particular such an approach is useful if no IDL files are used or have been created automatically by the IDE as it is in the case of VisualBasic.
- *Registry*: COM+ allows the configuration of component behavior at deployment time such as transaction semantic and security settings. This information gives a rough estimate about deployment specific architectural characteristics of COM+ components and complements the source code and type library analysis results.

The tools we use in our analysis process comprise existing reverse engineering tools such as Imagix-4D [7] (C/C++) and SourceNavigator [23] (VisualBasic) for parsing source code and visualizing results, Revealer [19] and grep for matching source code patterns that indicate COM+ characteristics in method implementations, and our Component Inspector tool for extracting meta data and configuration data from type libraries and the Windows Registry.

The complete analysis process, its results, and tools used in each step are depicted in Figure 1. The process starts with the identification of components in the source code of the client application (i.e. presentation layer). Subsequently, all components identified are used for meta data, configuration, and source code analysis. If any additional components are identified the process is applied iteratively for these new components.

Source code, type libraries, and the Windows Registry are the basic information sources for our analysis process. Each analysis step also uses the results produced by the previous step to extract and generate the data describing a COM+ component (i.e. interfaces, methods, deployment info, architectural characteristics). In this way each iteration reveals information about new identified COM+ components until all components of a software system have been analyzed. A detailed description of the four major phases of our analysis process is given in the following sections.

3.1. Component identification

For analyzing a three-tiered software system that uses COM+ components we start with the client application (i.e.

presentation layer) and determine all COM+ components that are used by it. To obtain a list of these components we investigate the client application's source code in which we basically focus on those portions of code that instantiate COM+ components. Programming frameworks for COM+ development provide specific statements for instantiation. Figure 2 shows an example for VisualC++ with ActiveX Template Library (ATL), VisualBasic, and Active Server Pages (ASP).

Each COM+ component can be identified by a *program identifier*. Therefore, when instantiating a COM+ component a program identifier has to be specified as an argument of the create statement. All three statements in Figure 2 create an instance of the component `CustomerC` whose program identifier is `M.CustomerC`. The standard format of a program identifier is

<Vendor>.<Component>.<Version>

Often the version number is omitted if there exists only one version of the COM+ component. To match instantiation statements we apply our lexical analysis tool Revealer [19] that facilitates the specification and extraction of simple source code patterns. The output of Revealer is a list of program identifiers along with their match location in source files. They indicate all COM+ components that are directly accessed by the client. Other components that are accessed indirectly by the client are determined in the same way by analyzing the source code of the components already identified.

```
// Visual C++
pObjCustomer.CoCreateInstance
  (__uuidof(M.CustomerC));

' Visual Basic
Set objCustomer = CreateObject
  ("M.CustomerC")

' Active Server Pages (ASP)
Set objCustomer = Server.CreateObject
  ("M.CustomerC")
```

Figure 2. Code snippets describing the instantiation of COM+ components in VisualC++, VisualBasic, and ASP

Program identifiers provide the starting points for the next analysis steps that are concerned with the extraction of meta and configuration data of each COM+ component. Therefore we developed the Component Inspector that given a program identifier finds the corresponding COM+ component, loads its type library and extracts the meta data, and retrieves its deployment data. A more detailed description of these two analysis steps is provided by the next two sections.

3.2. Meta data analysis

Component models provide meta data to aid system integration of components developed independently. Meta data contains descriptions about external visible interfaces and the corresponding methods that are exposed externally to clients.

COM+ stores meta data in type libraries. These libraries are either stored in additional files with suffix *.tlb* or directly placed in the component's image file [2]. In both cases the COM+ API function `LoadTypeLib` may be used to load a type library. These API function returns a COM+ component that implements the predefined `ITypeLib` interface and provides methods to retrieve the `ITypeInfo` interface that contains necessary functionality to access COM+ meta data about interfaces, methods, parameters, user-defined types, enumerations and all names used for defining this meta data.

If the location of the type library file is unknown but the component is installed on the system and supports the `IProvideClassInfo` interface or the `IDispatch` interface a reference to `ITypeInfo` can be retrieved by instantiating the component, retrieving either `IProvideClassInfo` or `IDispatch` and retrieving the interface pointer with a method provided by these interfaces.

A different means to get a reference to a type library is by searching the Windows Registry. The Registry stores the place where to find a type library in a key retrievable by the class identifier.

3.3. Configuration analysis

Some information of component based systems is not available during development time but is made available only at *deployment time*. This information describes transaction semantics or security settings that are dependent on the final installation of a system. Nevertheless, this information complements the information gained by meta data and source code analysis. Although this configuration settings are specific to a particular component deployment they give a rough estimate about the transaction and security architecture.

Distributed component models such as COM+ and EJB rely on role-based security. That means operating system users belong to certain roles and further security settings are based solely on these roles. COM+ allows fine-grained control over the realized roles, the mappings of operating system users to component roles, and the security settings of roles to whole components, to component interfaces and

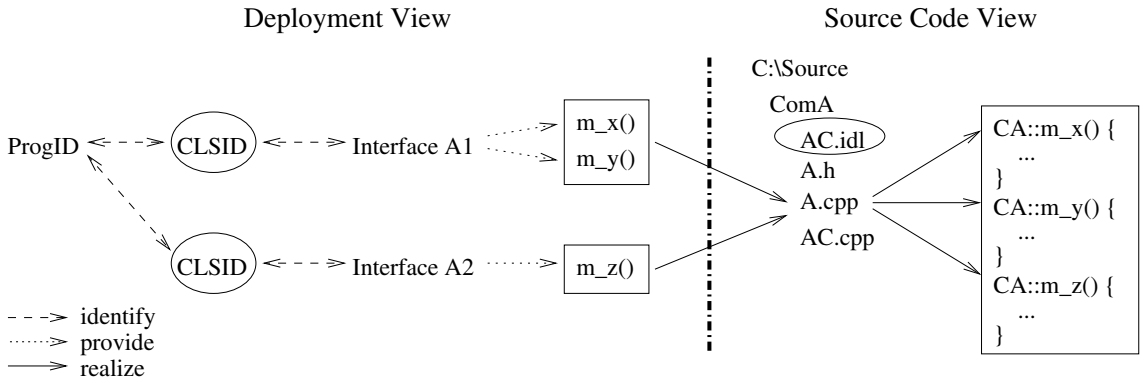

Figure 3. Mapping between COM+ component meta data and source code.

to interface methods. However, it is unlikely that role names will be defined completely different for each installation site. If roles are used within a component's code to realize fine-grained security control, the names of the roles are already fixed. The COM+ `COMAdminCatalog` component that ships with Windows 2000 and Windows XP provides interfaces to extract these data. We collect the security information that is configurable by COM+ for further processing.

Declarative transaction settings can be retrieved, too. Although the settings are restricted to if a transaction is supported, if a method has to be executed within an existing transaction, or if a new transaction shall be started on each method invocation of a component, this information shows a rough picture where transactions are necessary. As for security these settings are not completely independent of the implementation. In COM+ based web applications transactions are often used when updating the data storage. Hence, the methods involved in transactions will either lead to calls to a database or to the COM+ methods `SetAbort` and `SetComplete`. The COM+ `COMAdminCatalog` component is used to extract transactional settings to complement information gained by source code analysis.

Configuration information is used by the Component Inspector to extend the information collected during meta data analysis.

3.4. Source code analysis

The goal of the source code analysis is to provide a mapping between COM+ component interfaces and their implementations on the method level as well as a detailed characterization of each method with respect to relevant architectural characteristics. As a preliminary step of this analysis phase we use Imagix-4D and SourceNavigator to parse the source code of components. The generated source models are stored in the Imagix-4D database.

Figure 3 demonstrates the basic mapping of a COM+ component's meta data and its implementation. Each COM+ class is indicated by a unique class identifier. Therefore to create a mapping between a component and its implementation we use the class identifier of a component and apply a simple grep-query to search for a source code directory containing files that specify the corresponding CLSID (e.g. A.idl). Concerning the mapping of methods to source code we query the source model with respect to source files that are contained in the component's directory and further contain the implementation of a particular method. For example to map the methods `m_x()`, `m_y()`, and `m_z()` we query the files of directory `ComA`.

After obtaining the source code links we analyze each method implementation and concentrate on the following two questions:

- Does the method implement and make use of transactions, security, persistency, and state (shared property) mechanisms?
- Which other components are instantiated and used, stored, and returned by the method?

Both questions are concerned with important architectural characteristics whose answers lead to a better understanding of the system's software architecture. Therefore, we query the extracted source models and additionally perform lexical-based analysis on source code information not considered by parsers. The targets of our queries are given by the COM+ component model itself that provides various services which can be accessed by specific statements. For example transactions are controlled by the Microsoft Transaction Server (MTS) and basically driven by the `SetComplete` and `SetAbort` statement. `SetComplete` is called if an object wants to indicate that it has completed successfully and votes to commit any transaction it is enlisted in. On the contrary, `SetAbort` is called if the object

Characteristic	Statement	Description
transactions	SetComplete	vote to commit transactions
	SetAbort	vote to abort transactions
	EnableCommit	object is ready for commit
	DisableCommit	object is not ready for commit
security	IsSecurityEnabled	check if role based security is enabled
	IsCallerInRole	check if caller is member of specified role
state	CreatePropertyGroup	create shared property group
	CreateProperty	create shared property by name
	CreatePropertyByPosition	create shared property by position
	Property	retrieve reference to shared property by name
	PropertyByPosition	retrieve reference to shared property by position
persistency (ADO)	Open	open an ActiveX Data Objects (ADO) connection
	Execute	execute SQL statement
	SQL	select, insert, update, delete statement
create dependency	CreateInstance	create a single object
	CoCreateInstance	create a single object
	QueryInterface	get pointer to interface of an object currently accessed by client
error handling	AtlReportError	provide error information

Table 1. VisualC++ statements indicating architectural characteristics of COM+ components.

votes to abort its transaction. Finding such statements in a method implementation simply indicates corresponding architectural characteristics and component interaction. Table 1 shows an excerpt of important COM+ related VisualC++ statements that we currently take into account. Regarding VisualBasic or ASP we created similar lists.

Besides the create dependency we also consider statements that represent use relations between components such as component A accesses services (e.g. calls methods) of component B.

For each COM+ component the source code analysis tool stepwise fills the data model with the name of the corresponding source code directory, the name of files implementing the interfaces, and information about the interface method implementations. Latter comprises the source code locations (file names plus line numbers) and a list of characterizing statements including the match location (line number) and the architectural characteristic affected by it.

To get a complete picture of a COM+ component we combine source code information with meta data and configuration details. Such a combination is necessary because COM+ also supports the use of transactions and security settings that have no counterpart in the source code. For example, database access frameworks that are prepared to automatically make use of transactions such as ActiveX Data Objects (ADO) [21] or OLE DB [15] need no specific transaction handling in components. Although, the component that invokes the database access has no control over the transaction COM+ allows configuration of transaction settings. Hence, using only source code analysis cannot generate a complete picture of the architectural characteristics.

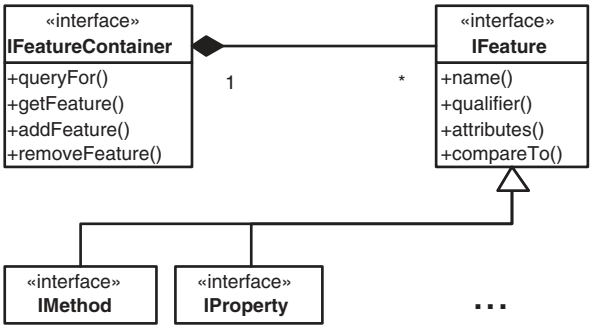

Figure 4. Feature Container Design

3.5. Data model

All data collected during the different analysis steps will be stored in one data model. We reused the internal representation for components that has been implemented for the Vienna Component Framework (VCF) [18]. This Java based framework provides a generic bridge facility for different component models. It provides an abstract model that allows the administration of component features such as methods, properties and events in a tree-like data structure. Figure 4 shows a class diagram of feature containers and features constituting the data structure.

The VCF uses feature containers to organize sets of component features and attributes to store associated detail data

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

Characteristic	Attribute class	Description	Analysis phase
transactions	DeclarativeTransactionSupport	Transaction type	configuration
	TransactionIsolationLevel	Isolation level	configuration
	Synchronization	Synchronization level	configuration
	ProgrammaticTransaction	Code statement & source code location	source code
security	AccessCheckEnabled	Role-based access check enabled	configuration
	CheckAccessCheckEnabled	Determine if based access check is enabled	source code
	TotalRoles	All roles and users assigned to a component	configuration
	Roles	Roles allowed to access a feature	configuration
	CallerInRole	Source code location	source code
dependency	CreateInstance	Create a component instance	source code
	QueryInterface	Query a component interface	source code
	Call	Call a method of another component instance	source code

Table 2. Attribute classes representing component characteristics.

to features. Particular features, called hierarchical features, administrate their own feature containers and in this way allow for a flexible hierarchical composition of features. With respect to the output data generated by our analysis process the feature data structure is filled top-down from components, followed by interfaces down to methods, properties, and events. Latter are represented as leaf nodes in this tree. Each feature can have an unlimited number of associated attributes. While the feature types are predefined by VCF, attributes can be arbitrary Java objects. This allowed us to extend VCF with custom attributes when needed.

During meta data analysis a feature container is created for each component that is investigated and tree nodes are created for each COM+ interface, method and property, and inserted into the tree. All additional characteristics that are found by configuration and source code analysis are inserted into the container as attributes associated to already existing features. Table 2 shows some of the attribute classes we have implemented to represent different component characteristics and their origin.

The VCF provides predefined data types and methods to query feature containers for elements with certain characteristics. It is possible to find all features with particular attribute types or attribute values. Since we added new attribute classes it was possible to search for all methods that interact with one specific method of another component, or all methods that are involved in transactions and contain calls to database routines.

4. Case study

To demonstrate our approach we applied the analysis tools to the Island Hopper application, a sample application of about 10 KLOC described in [10] to illustrate design and implementation concepts of COM+ based application development. The software architecture of Island Hopper is three-tiered in which the presentation logic is implemented using Active Server Pages (ASP) and the business and data access logic using VisualC++ and VisualBasic. With regard to a component view all components of the business and data access tier are COM+ components whose services are accessed top-down from the client application. We installed all sixteen components (10 VisualBasic and 6 VisualC++) of the Island Hopper application and their sources and performed our analysis steps as follows:

1. We use Revealer to query the ASP-files of the client application and match COM+ object instantiation statements (i.e. `CreateObject(''ProgID'')`). Revealer outputs a list consisting of six different program identifiers whereas two of them indicate Windows components handling database access (ADODB.Resultset) and scripting (Scripting.FileSystemObject).

2. Using the program identifiers we apply our COM+ inspection tool to extract the meta data of COM+ components as described in Section 3.2

3. We further use the Component Inspector to obtain the configuration data. Table 3 shows an excerpt of the results collected during the first three steps containing the program identifiers, interfaces and methods of two investigated COM+ components `M.CustomerC` and `M.ProductC`.

4. We use Imagix-4D (C/C++) and SourceNavigator (VisualBasic) to parse the source code of the Island Hopper components that we already have identified so far. For each interface of these components we investigate the method implementations with respect to the recovered create dependencies to determine the remaining COM+ components indirectly used by the client application until we get the "transitive closure" of all

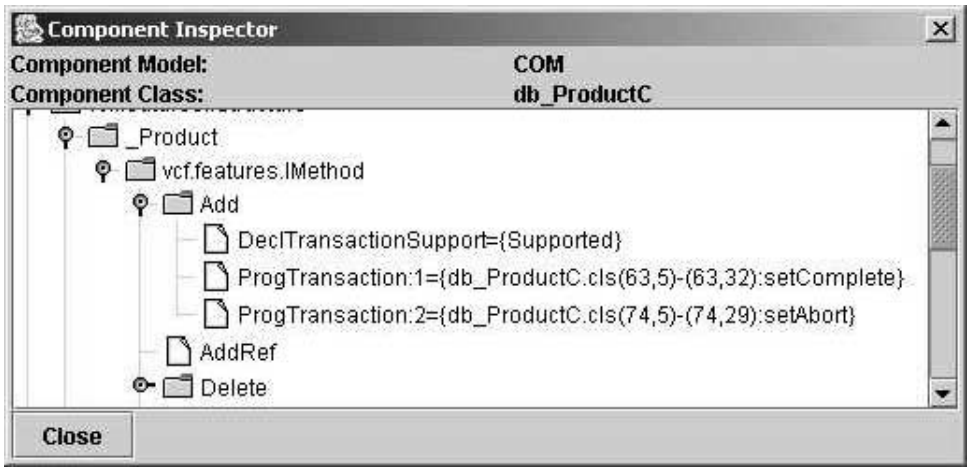

Figure 5. Component Inspector showing features of the db_ProductC component.

COM+ components of the Island Hopper application. For each new identified component we repeat the analysis steps two and three to obtain their meta data and configuration details.

Based on the set of all COM+ components we analyze all method implementations with respect to statements indicating architectural characteristics. We extract such indicators by querying the source models stored in the Imagix-4D database and the corresponding source code. All results generated by our analysis tools are stored in feature containers as described in Section 3.5.

ProgID	Interface	Method
M.CustomerC	ICustomer	Validate()
		Add()
		GetByEmail()
	IChange	Add()
		Update()
		Delete()
	ILookup	GetByID()
		ListByLastName()
		GetByEmail()
M.ProductC	IProduct	Add()
		Delete()
		Update()
		GetByID()
		GetUnitPrice()
		ListByDesc()

Table 3. Meta data of two COM+ components identified and extracted during the first three steps of our analysis process.

Through the combination of the data extracted from all three different information sources we obtain a meaningful picture of the Island Hopper application that provides a basis for further architectural reasoning. We have provided a graphical user interface for the Component Inspector to quickly navigate through components, interfaces, methods, architectural characteristics, and their attributes. Through the source code mapping we are also able to browse those portions of code that implement certain characteristics. Figure 5 represents the Component Inspector showing a subset of features extracted for component `db_ProductC`. In particular, it shows the declarative and programmatic transaction properties implemented by the method `add()`. For latter properties also the location (file name and line numbers) of the corresponding source code statements are listed. This location information is exploited to navigate to text windows showing the corresponding source code.

Besides the usefulness of our approach the case study also indicated some problems that may influence the accuracy of results produced by our approach. For example it is crucial to configure the C/C++ parser of Imagix-4D towards the emulation of the VisualC++ parser to get accurate source models. Basically this concerns the specification of preprocessor statements and system header files to include because on the one hand the source model database can get large when analyzing system header files is enabled, but on the other hand you loose important source model information such as methods and data types predefined by the development environment. For our case study we adjusted this setting to include all necessary header files.

Although Imagix-4D emulates the VisualC++ parser it cannot fully handle C++ templates that are heavily used within Microsoft's COM/COM+ programming models. Basically, the problem is in the instantiation of nested tem-

plates and causes the loss of type information. Therefore, some workaround is needed to determine for example the COM+ instance whose method is called.

Another problem appeared during meta data analysis concerning the use of the generic COM/COM+ `VARIANT` data type and untyped pointer arguments because their actual types can only be determined at run-time. Fortunately source code analysis can provide more accurate typing information in many cases.

5. Related work

Software components are of primary concerns in the field of reverse engineering and there exists several tools and techniques that address the extraction of software components from available information (e.g. source code). However, these approaches don't take into account the information and architectural characteristics that are implicated by a particular application domain such as COM/COM+ component-based three-tiered applications. With respect to that our approach is novel.

For example Koschke [13] describes an approach that focuses on the detection of atomic components by integrating the user into the detection cycle. Atomic components are groupings of subprograms, types, and global variables, and can be viewed as cohesive logical modules. In contrast, we view components as deployable units that are self-contained.

Sneed et al. [22] describe a tool capable of analyzing very large applications. It uses a relational database for storing both the requirements specification and also the created implementation model. A mapping between records that represent particular requirements and those that represent implementation models is established in the database, too. Subsequently the database can be queried by SQL statements to find out about various system properties.

Andrews et al. [1] identify Commercial-of-the-Shelf (COTS) component comprehension as a specialized activity within software comprehension. They build a combined comprehension model for COTS components based on a domain model, a situation model, and a program model and evaluate how different component based software development approaches fit with this model. Unlike our approach they focus on abstract comprehension models without providing a concrete implementation.

Korel [11] deals with COTS components, too. He treats components as black-box entities and uses interface probing as primary tool for obtaining component properties. To realize interface probing a developer has to design a set of test cases that are executed on the component. The results can be evaluated and interface probing can be applied iteratively. While black-box understanding methods can be applied when no source code is available they are limited with respect to the number of properties they can extract.

Java applets are specific Java components that can be started in web browsers. Korn et al. have developed Chava that supports the analysis of Java applets [12]. They combine source code analysis with bytecode analysis techniques. Similar to our meta data analysis bytecode analysis can reveal some information about Java classes but cannot show the full picture since the Java compiler removes information.

6. Conclusions

Software systems are more and more built from components that follow a certain component framework such as COM/COM+, EJB, ASP, or JSP. Maintaining and evolving such systems, therefore, offers new challenges for program analysis and understanding.

We have developed an approach that specifically supports analysis and understanding of COM/COM+ component-based systems. It provides semi-automatic analysis techniques to investigate architectural characteristics of COM+ components such as transactions, security, persistence, error handling, or component dependency. As a result, the engineer receives descriptions of component attributes and inter-component relationships that enable him to faster understand components and their interactions and to guide him in further evolving the system.

Our approach is implemented in the Component Inspector tool and combines static analysis with standard reverse engineering tools. Different analysis steps use specific tools (e.g. Imagix-4D, Revealer, or SourceNavigator) and collect the necessary information and present it on the Component Inspector. Since different levels of abstractions are considered and linked via the different representations of the recovered information (meta-data, source code, etc.) the approach allows the engineer to navigate from architectural characteristics (such as transactional or persistency aspects) all the way down to the corresponding source code. We have discussed the technical foundations of our approach for COM/COM+ components and evaluated it with a case study - the Island Hopper application.

Future work will concentrate on providing the semi-automatic transformation to move from one component model to another (e.g. from COM+ to EJB) thereby using the previously developed Vienna Component Framework and extending the Component Inspector tool.

Acknowledgements

We are grateful to the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT) and the European Commission for funding our work under EUREKA 2023/ITEA-ip00004 'from Concept to Application

in system-Family Engineering (CAFÉ)' and the IST Project 1999-14191 'Easy Composition in Future Generation Component Systems (EASYCOMP)'.

We also would like to thank Michael Fischer for fruitful discussions on the COM+ analysis process and his comments on the paper, as well as the anonymous reviewers for their feedback.

References

[1] A. Andrews, S. Ghosh, and E. M. Choi. A model for understanding software components. In *Proc. of the International Conference on Software Maintenance*, pages 359–368, Montreal, Canada, October 2002. IEEE Computer Society Press.

[2] K. Brockschmidt. *Inside OLE*. Microsoft Press, second edition, 1995.

[3] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002.

[4] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[5] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[6] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[7] Imagix-4d 4.1. http://www.imagix.com, December 2002.

[8] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, Reading, Mass. and London, 2000.

[9] R. Kazman and S. J. Carriére. Playing detective: Reconstructing software architecture from available evidence. *Journal of automated Software Engineering*, 6(2):107–138, April 1999.

[10] M. Kirtland. *Designing Component-Based Applications*. Microsoft Press, 1999.

[11] B. Korel. Black-box understanding of cots components. In *Seventh International Workshop on Program Comprehension*, pages 226–233. IEEE Computer Society Press, May 1999.

[12] J. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. of the 6th Working Conference on Reverse Engineering*, pages 314–325. IEEE Computer Society Press, Oct. 1999.

[13] R. Koschke. An incremental semi-automatic method for component recovery. In *Sixth Working Conference on Reverse Engineering*, pages 256–267. IEEE Computer Society Press, Oct. 1999.

[14] S. M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys (CSUR)*, 30(1):3–27, 1998.

[15] *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*. Microsoft Press, 1998.

[16] Microsoft Corporation. *The Component Object Model Specification*, 1995.

[17] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., first edition, June 1999.

[18] J. Oberleitner, T. Gschwind, and M. Jazayeri. Vienna component framework: enabling composition across component models. In *Proceedings of the 25th international conference on Software engineering (ICSE)*. ACM Press, 2003. to appear.

[19] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proc. of the 9th Working Conference on Reverse Engineering*, pages 170–178, Richmond, Virginia, October 2002. IEEE Computer Society Press.

[20] M. Pinzger and H. Gall. Pattern-supported architecture recovery. In *Proc. of the 10th International Workshop on Program Comprehension*, pages 53–61, Paris, France, June 2002. IEEE Computer Society Press.

[21] D. Sceppa. *Programming ADO*. Microsoft Press, 2000.

[22] H. M. Sneed and T. Dombovari. Comprehending a complex, distributed, object-oriented software system: a report from the field. In *Seventh International Workshop on Program Comprehension*, pages 218–225. IEEE Computer Society Press, May 1999.

[23] Source-navigator 5.1. http://sourcenav.sourceforge.net, June 2002.

[24] Sun Microsystems. *Enterprise JavaBeans*, 2.0 edition, Oct. 2000. http://java.sun.com/products/ejb.

[25] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[26] A. Troelson. *COM and .NET Interoperability*. APress, Apr. 2002.