

Describing Dependencies in Component Access Points

Marlon E. R. Vieira

Marcio S. Dias

Debra J. Richardson

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{mvieira, mdias, djr}@ics.uci.edu

ABSTRACT

Developing component-based systems is problematical without understanding potential component dependencies. A significant underlying challenge is making available ways to determine and represent those dependencies and mechanisms to deal with them. This paper discusses issues related to component dependencies and introduces an approach to describe what can happen (in term of actions/dependencies) after a particular component's access points (services) are called.

Keywords

Software components, maintenance, dependence analysis.

1 INTRODUCTION

There is continuing interest in using components to build software systems. Components can be viewed as self-contained software entities that interact through open interfaces. They may be developed in-house or made available by a third party, often commercial-off-the-shelf (COTS). The component-based development paradigm [1] promises better quality and productivity, in part by addressing reuse and evolution of functionality over time. Unfortunately, this paradigm hasn't reached yet its full promise; in fact, it is presenting some problems more difficult than the "traditional" development paradigm [2]. Several difficult technical problems remain to be explored and resolved in this field, many resulting from component heterogeneity and the lack of behavioral information about deployed components. Heterogeneity [3] takes place due to the fact that different components can be programmed in different programming languages and for different hardware platforms. Components may also be provided as "black-box" by different, possibly competing suppliers, often deficient in supplied information.

A range of technologies support component-based development (e.g. OMG's Corba, Microsoft's COM/COM+, Sun's (Enterprise) Java Beans). Those

technologies implement open standards offering an infrastructure of services to create, assemble and execute software components. In addition, they embody mechanisms to solve some problems with respect to component heterogeneity, allowing better interoperability. In spite of the important issues in component development that these technologies address, they don't support precise component description. In general, components are deployed with an informal description of services provided, together with the signature of the methods that invoke those services. This specification does not provide enough information for reasoning about component interactions beyond the compatibility of their interface, consequently restricting how well system properties can be predicted and verified.

Aside from problems associated with creating, deploying and composing components, there are questions related to verification/validation of component-based system. It is not clear, for instance, how the behavior of such systems can be verified and controlled in the presence of COTS components. More specifically: (i) how is it determined whether or not component behavior (or misbehavior) can affect other components and the system as a whole? (ii) what are the factors that determine cooperation and dependence among components, and (iii) under what circumstances can problems arise in those relations? Maintenance concerns also play a central role in component-based development. Adding, removing or modifying components requires a full cycle of verification/validation. It is difficult to perform those tasks without understanding potential component dependencies. Making available ways to locate and represent dependencies among components, along with mechanisms for managing those dependencies, present significant challenges for component-based development.

The goal of the research reported here is to investigate the sources of component dependencies and to develop an approach to describe them. Our approach relies on description of what can take place after a particular component's access points (services) are called. Typically, an interface specification doesn't stipulate what happens (in terms of possible actions/dependencies) when one of its methods is invoked. For instance, a component can request

a service from another component before a possible call-back to the originating component request.

Dependence analysis is a fundamental static analysis technique that has been useful for understanding [6], testing [7], debugging [8], and maintaining [9] software systems. Considerable work has been done in the study and use of dependence analysis in the context of traditional programming languages, generally looking for correlation among variables and statements. Likewise, there is some relevant work that applies dependence analysis in the context of software requirement specification [10] and software architecture specification [11][12]. Specification dependence analysis is a generalization of program dependence analysis since a programming language can be viewed as an executable specification language.

Based on the specification of the component's access points, dependence analysis can be performed to predict system dependencies. In our approach, those dependencies are presented as relationships among different services and parameters of those services. We believe that this information is able to provide helpful subsidies to carry out maintenance tasks as well as verification/validation tasks.

2 COMPONENT DEPENDENCY ISSUES

Informally, dependence among components can be defined as the reliance of a component on other(s) to support a specific functionality or configuration. In highly structured environments, like component-based systems, units of computations (e.g. components) communicate and share information in order to provide system functionalities. Components are regularly composed for the purpose of offering more abstract services in a system. This composition creates interactions that promote dependencies among the components. Frequently, system functionalities are not solely encapsulated within one component. Therefore, changing a component can affect that composite functionality, which is reflected in different components. In addition, replacing a new version of a specific component might involve replacing the components on which it depends, in order to preserve a specific system's functionality. The key point to analyzing those aspects is the knowledge about possible component relations and the dependencies among them.

One example of dependence is when a group of components depend on each other to supply a complex system functionality (e.g. a logon functionality of an on-line bank system might involve executing services of different components for authentication, security, information retrieval and provide specialized functionalities). Another example is when a specific component relies on other(s) to obtain service(s) necessary for its own functionality (e.g. a spell-checking component may require the services of a dictionary component to perform its tasks).

Besides dependencies on other component(s), a specific component can also depend on a particular

hardware/software platform and/or version. Szyperki [4] states the existence of components' explicit *context dependencies*; those dependencies are constituted by what a system, with which the component is going to be integrated, needs to provide to fill the component's needs. Kon and Campbell explored component dependencies [5] and define two distinct kinds of dependencies related with run-time evolution: (1) *prerequisites* - requirements for loading an inert component into the runtime system and (2) *dynamic dependencies* - relations between loaded components in a running system.

3 COMPONENT DESCRIPTION

The concept of interfaces is an important issue in component-based development. Components are expressed in terms of externally visible interfaces and semantics, without the implementation [13]. This view disconnects the information regarding *what* a component does from *how* it does it. An interface defines what services a component provides and requires in order to be used in a system. The provided services identify the component's *access points* for its clients.

Interfaces can be defined at different levels of precision. A superior precision allows more effective component composition and conformance checking against its specification. The exact way an interface is specified depends on the nature of the component; generally, however, a component interface is described by the signature of its operations and required calling conventions. *Interface Definition Languages* (IDLs) are frequently used to describe distributed software component interfaces, providing a programming neutral way to describe component services. For instance, the CORBA IDL [16], standardized by the OMG, defines interfaces as sets of synchronous request/reply operations or asynchronous one-way messages. A notable limitation of current IDLs is that they only provide support for describing the names and type signatures of the component's attributes and operations, ignoring aspects related to behavior. Behavioral aspects are important to analyzing system properties and to carrying out testing and maintenance activities. Some approaches [14][15] try to fill this gap by specifying component behavior with assertions (pre and post-conditions). Preconditions express the properties that must hold whenever an operation is called; if it is violated, the caller of the service is responsible. Postconditions describe the properties that an operation guarantees when it returns; if it fails there is a problem in the service (component) itself.

The focus of our research presented here is on the relationship among components, describing component interaction structure. This specification makes clear if a component needs some service in the system to be available in order to perform the functionality exposed by an access point and shows the composition of services among components. The main idea behind this is quite simply: Having provided a description of what can happen

after a service is solicited, it will then be easier to predict the dependencies of a system composed of those components, thereby subsidizing system analysis.

4 DESCRIBING COMPONENTS ACCESS POINTS: AN OVERVIEW

It has long been recognized that the best way to describe the semantics of a software component is abstractly and formally [17]. We believe it would be useful to have precise and high-level documentation related to component dependencies as well. The approach we take in providing this mechanism is a specification language to describe component access points (or services). That description can be deployed with the component and used to support analysis tasks. The prime reason for creating a new description instead of extending an IDL with assertions (and/or new keywords) is to enable support for (semi) automatic extraction of this description from the source code, thus providing a higher level of abstraction than what is implemented. Further, we want to maintain a more general approach (not coupled with a specific IDL), allowing compatibility with different technologies. The specification of component access points is intended to complement an IDL and not serve as a substitute.

In our approach, we define component access points, which consist of a sequence of one or more actions. The description consists of (i) a header, specifying the operation's signature, and (ii) a body, describing the operation's behavior (possible actions) after it is invoked. Since an operation in our description executes as a sequence of one or more atomic actions, we specify its behavior not with a single pre-condition/post-condition pair, but rather, with a sequence of specifications, each describing the behavior of one atomic action. Each atomic action has a *when* clause and an additional *effects*. A list of action identifiers declares the sequencing of the actions. An action specification consists of the action's name and one or more action bodies, each containing:

- *when* clause, which states the condition that allows the action to take place. If the condition for an action is not met, the action will not be performed, but with the execution of other actions, their effects may allow its when-condition to be met later. The default for the when clause is true, which means no restriction to execute the action.
- *modifies* clause, which specifies if parameters may change as a result of the action. This clause defines that only the listed parameters may change; all other parameters remain unchanged. A keyword *define* inside the modifies clause establishes a parameter that is first defined during that action.
- *effects* clause, which describe the results of the action. A parameter listed in the modifies clause may either be bound to a subjective value defined by the keyword *arbitrary*, or if the effect is specified, bound to a specific value that is the consequence of the action. It is assumed

that any parameter not listed in the *effects* clause is bounded with an arbitrary value.

The *composition of* statement allows the specification of a sequence of atomic action. We define four basic types of atomic actions:

1. Actions dependent on services that are not provided by the component itself; consequently, the component that carries out that action relies on the system (typically another component) to provide the service specified by the action.
2. Internal component services that are not explicitly provided for clients; the description of an action of this type shares knowledge about the component's internal flow, which is normally hidden from clients (actions of this type can be omitted if the developer does not want to expose the component's internal structure).
3. Services that the component provides through its interface; this description also shares information about the component's internal flow.
4. Terminal action, which is defined by the keyword *operationReturn*; atomic actions always execute in some particular order, so the terminal action puts together the results of execution and ensures the operation return, in the case that here is a return.

Figure 1 presents a very simple example of component access point specification for a login service (operation) of a BankOnline component.

```

Component BankOnline
{
  [...]
  operation login ( String ID, int password ) returns (double balance)
  composition of verifyIdentity, getAccountBalance, operationReturn
  action verifyIdentity (String ID, int password) returns (int acc)
    when true
    modifies define (acc)
    effects (arbitrary or acc = null)
  action getAccountBalance (int acc) returns (double balance)
    when (acc <> null)
    modifies define (balance)
    effects (arbitrary)
  action operationReturn (int acc, double balance ) returns (double
balance)
    when (acc = null)
    modifies (balance)
    effects (balance= null)

    when (acc <> null)
    effects (arbitrary) //return balance
  [...]
}

```

Figure 1 - Login operation

When the login service is called the user identification is verified and his/her balance is returned, in the case of wrong identity the service returns null (exception). This specification consists of a composition of three actions. For the sake of brevity (and space limits), we do not illustrate the entire example; the *verifyIdentity* and *getAccountBalance* actions are related, respectively, to services provided by authentication component and

database component. Using the description provided for the diverse components access points, we can construct a dependence graph showing components in the nodes and actions in the edges.

5 CONCLUSION AND ONGOING WORK

The motivation that led us to explore component dependencies is the need to understand how system components interact and depend on each other during component-based development. The notion of context unawareness (independently deliverable entities) leads to the necessity that components be developed without embedded dependence on other components, but it does not mean that components have no dependencies on one another. The unawareness concept stresses the notion of no interference between components. An important point, however, is that they are projected to work together to achieve a solution. The interaction necessary to reach the solution results in a set of dependencies that forms an essential part of the problem to be solved. Describing and analyzing these dependencies have become an important part of software component research.

Describing what is supposed to happen after each component access point is activated allows *separation of concerns* in our approach, because only a small part of component behavior is specified at a time. Those specifications can be combined resulting in the overall component interface behavior (in terms of actions). From this information, analysis can determine if a component can achieve a behavior required for a particular context. Also, the conceptual model created with the description of possible components interaction (dependencies) in a system can be effectively used to support testing and maintenance activities. The combination of components access point description can give to the developer a broad idea of the system internal interaction network, helping, for instance, to determine if a change in one component can have effects in other component(s) or to chose which components need to be re-tested after a system functionality be modified.

We have argued in favor of a formal approach to specifying component access points, and have presented an overview of our specification language. The precise nature of the semantic foundation and of the language itself have not been fully defined here due to space limitations. Our research directions include improving the description language after exploring its use over different component technologies, such as (Enterprise) JavaBeans, COM/COM+ and CORBA. We also intend to provide better tool support (an assistant) to describe component access points. One significant and challenging point in our research is to extract high-level (precise) specifications from source code. We are evaluating "inverse engineering" techniques [18][19] to perform that task. Inverse engineering is based on a theory of program refinement and transformation, which is used as the basis for the development of a catalogue of powerful semantics-preserving transformations to discover specification for the program.

6 REFERENCES

- [1] A. W. Brown, Large Scale Component Based Development, Prentice Hall, 1st edition December, 2000.
- [2] A. Orso, M. J. Harrold, and D. Rosenblum, "Component Metadata for Software Engineering Tasks", In Proceedings of the 2nd International Workshop on Engineering Distributed Objects, Davis, CA, November 2, 3, 2000
- [3] D. S. Rosenblum, "Adequate testing of component based software", Technical Report 97-34, ICS, University of California, Irvine, august 1997
- [4] C. Szyperski, Component Software - Beyond Object-Oriented Programming, Addison-Wesley, 1998
- [5] F. Kon and R. Campbell, Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 8(1), pp. 26-36, January-March, 2000.
- [6] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing", In Proceedings of the Fourth Workshop on Program Comprehension, Berlin, Germany, March 1996.
- [7] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs", Conf. Record of the 20th Annual ACM SIGPLAN-SIGACT, Sym. of principles of Programming Languages, pp.384-396, ACM Press, 1993.
- [8] H. Agrawal, R. Demillo, and E. Spaord, "Debugging with Dynamic Slicing and backtracking", *Software Practice and Experience*, Vol.23, No.6, pp.589-616,1993.
- [9] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance", *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.
- [10] J. Chang and D. J. Richardson, Static and Dynamic Specification Slicing, *Fourth Irvine Symposium on Software*, April 1994.
- [11] J.A. Stafford, D.J. Richardson, and A. L. Wolf, Chaining: A Software Architecture Dependence Analysis Technique, *Technical Report CU-CS-845-97, University of Colorado*, September 1997.
- [12] J. Zhao, Using Dependence Analysis to Support Software Architecture Understanding, in M. Li (Ed.), *New Technologies on Computer Software*, pp.135-142, International Academic Publishers, September 1997.
- [13] T. Digre, "Business Object Component Architecture", *IEEE Computer*, Sept/Oct 1998, pp60-69
- [14] D. F. D'Souza and A. Wills, Objects, *Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.
- [15] S. Edwards, G. Shakir, M. Sitaraman, B.W. Weide, and J. Hollingsworth, A Framework for Detecting Interface Violations in Component-Based Software, *In Proceedings of the Int. Conference on Software Reuse*, June 1998
- [16] Object Management Group, The Common Object Request Broker: Architecture and Speciation, Version 2.0, July, 1996
- [17] A. Brown, From Component Infrastructure to Component-Based Development, 20 th ICSE International Workshop on Component-Based Software Engineering (Kyoto, Japan), 1998.
- [18] P.T.Breuer., *Inverse Engineering: The First Step*, REDO Document 2487-TN-PRG-1031, Programming Research Group, Oxford University, 1990.
- [19] M. Ward, Specifications from Source Code -- Alchemists' Dream or Practical Reality? 4th Reengineering Forum, September 19-21, 1994, Victoria, Canada