Component Metadata for Software Engineering Tasks

Alessandro Orso[1], Mary Jean Harrold[1], and David Rosenblum[2]

[1] College of Computing
Georgia Institute of Technology
{orso,harrold}@cc.gatech.edu
[2] Information and Computer Science
University of California, Irvine
dsr@ics.uci.edu

Abstract. This paper presents a framework that lets a component developer provide a component user with different kinds of information, depending on the specific context and needs. The framework is based on presenting this information in the form of metadata. *Metadata* describe static and dynamic aspects of the component, can be accessed by the user, and can be used for different tasks throughout the software engineering lifecycle. The framework is defined in a general way, so that the metadata can be easily extended if new types of data have to be provided. In our approach, we define a unique format and a unique tag for each kind of metadata provided. The tag lets the user of the component both treat the information provided as metadata in the correct way and query for a specific piece of information. We motivate the untapped potential of component metadata by showing the need for metadata in the context of testing and analysis of distributed component-based systems, and introduce our framework with the help of an example. We sketch a possible scenario consisting of an application developer who wants to perform two different software engineering tasks on her application: generating self-checking code and program slicing.

Keywords: Components, component-based systems, distributed components, metadata.

1 Introduction

In recent years, component-based software technologies have been increasingly considered as necessary for creating, testing, and maintaining the vastly more complex software of the future. Components have the potential to lower the development effort, speed up the development process, leverage other developers' efforts, and decrease maintenance costs. Unfortunately, despite their compelling potential, software components have yet to show their full promise as a software engineering solution, and are in fact making some problems more difficult. The presence of externally-developed components within a system introduces new challenges for software-engineering activities. Researchers have reported many problems with the use of software components, including difficulty in locating the code responsible for given program behaviors [4], hidden dependences among components [4, 5], hidden interfaces that raise security concerns[12, 17], reduced testability [18], and difficulties in program understanding [4]. The use of components in a distributed environment makes all the above problems even more difficult, due to the nature of distributed systems. In fact, distributed systems (1) generally use a middleware, which complicates the interactions among components, and (2) involve components that have a higher inherent complexity (e.g., components in e-commerce applications that embody complex business logic and are not just simple GUI buttons).

Several of the above problems are due to the lack of information about components that are not internally developed. Consider an application developer who wishes to use a particular component by incorporating it into her application, either by using it remotely over a network or by interacting with it through middleware such as CORBA [6]. The application developer typically has only primary interface information supporting the invocation of component functions. In particular, she has no source code, no reliability or safety information, no information related to validation, no information about dependences that could help her evaluate impacts of the change, and possibly not even full disclosure of interfaces and aspects of component behavior. When the task to be performed is the integration of the component, information about the component interface and its customizable properties can be all that is needed. Other software engineering tasks, however, require additional information to be performed on a component-based system.

In this paper, we present a framework that lets the component developer provide the component user with different kinds of information, depending on the specific context and needs. The framework is based on presenting this information in the form of metadata. *Metadata* describe static and dynamic aspects of the component, can be accessed by the user, and can be used for different tasks. The idea of providing additional data together with a component is not new: It is a common feature of many existing component models, albeit a feature that provides relatively limited functionality. In fact, the solutions provided so far by existing component models are tailored to a specific kind of information and lack generality. To date, no one has explored metadata as a general mechanism for aiding software engineering tasks, such as analysis and testing, in the presence of components.

The framework that we propose is defined in a general way, so that the metadata can be easily extended to support new types of data. In our approach, we define a unique format and a unique tag for each kind of metadata provided. The tag lets the user of the component both treat the information provided as metadata in the correct way and query for a specific piece of information. Because the size and complexity of common component-based software applications are constantly growing, there is an actual need for automated tools to develop, integrate, analyze, and test such applications. Several aspects of the framework that we propose can be easily automated through tools. Due to the way the metadata are defined, tools can be implemented that support both the developer who has to associate some metadata with his component and the user who wants to retrieve the metadata for a component she is integrating into her system.

We show the need for metadata in the context of analysis and testing of distributed component-based systems, and introduce our framework with the help of an example. We sketch a possible scenario consisting of an application developer who wants to perform two different software engineering tasks on her application. The first task is in the context of self-checking code. In this case, the metadata needed to accomplish the task consist of pre-conditions and post-conditions for the different functions provided by the component, and invariants for the component itself. This information is used to implement a checking mechanism for calls to the component. The second task is related to slicing. In this case, the metadata that the developer needs to perform the analysis consist of summary information for the component's functions. Summary information is used to improve the precision of the slices involving one or more calls to the component, which would otherwise be computed making worst-case assumptions about the behavior of the functions invoked.

The rest of the paper is organized as follows. Section 2 provides some background on components and component-based applications. Section 3 presents the motivating example. Section 4 introduces the metadata framework and shows two possible uses of metadata. Section 5 illustrates a possible implementation of the framework for metadata. Finally, Section 6 draws some conclusions and sketches future research directions.

2 Background

This section provides a definition of the terms "component" and "component-based system," introduces the main technologies supporting component-based programming, and illustrates the different roles played by developers and users of components.

2.1 Components

Although there is broad agreement on the meaning of the terms "component" and "component-based" systems, different authors have used different interpretations of these terms. Therefore, we still lack a unique and precise definition of a component. Brown and Wallnau [2], define a component as "a replaceable software unit with a set of contractually-specified interfaces and explicit context dependences only." Lewis [10] defines a component-based system as "a software system composed primarily of components: modules that encapsulate both data and functionality and are configurable through parameters at run-time." Szyperski [16] says, in a more general way, that "components are binary units of independent production, acquisition, and deployment that interact to form a functioning system."

In this paper, we view a *component* as a system or a subsystem developed by one organization and deployed by one or more other organizations, possibly in different application domains. A component is open (i.e., it can be either extended or combined with other components) and closed (i.e., it can be considered and treated as a stand-alone entity) at the same time. According to this definition, several examples of components can be provided: a class or a set of cooperating classes with a clearly-defined interface; a library of functions in any procedural language; and an application providing an API such that its features can be accessed by external applications. In our view, a *component-based* system consists of three parts: the user application, the components, and the infrastructure (often called middleware) that provides communication channels between the user application and the components. The user application communicates with components through their interfaces. The communication infrastructure maps the interfaces of the user application to the interfaces of the components.

2.2 Component Technologies

Researchers have been investigating the use of components and component-based systems for a number of years. McIlroy first introduced the idea of components as a solution to the software crisis in 1968 [13]. Although the idea of components has been around for some time, only in the last few years has component technology become mature enough to be effectively used. Today, several component models, component frameworks, middleware, design tools, and composition tools are available, which allow for successful exploitation of the component technology, and support true component-based development to build real-world applications.

The most widespread standards available today for component models are the CORBA Component Model [6], COM+ and ActiveX [3], and Enterprise JavaBeans [7]. The CORBA Component Model, developed by the Object Management Group, is a server-side standard that lets developers build applications out of components written in different languages, running on different platforms, and in a distributed environment. COM+, OLE, and ActiveX, developed by Microsoft, provide a binary standard that can be used to define distributed components in terms of the interface they provide. The Enterprise JavaBeans technology, created by Sun Microsystems, is a server-side component architecture that enables rapid development of versatile, reusable, and portable applications, whose business logic is implemented by JavaBeans components [1]. Although the example used in this paper is written in Java and uses JavaBeans components, the approach that we propose is not constrained by any specific component model and can be applied to any of the above three standards.

2.3 Separation of Concerns

The issues that arise in the context of component-based systems can be viewed from two perspectives: the component developer perspective and the component user (application developer)[1] perspective. These two actors have different knowledge, understanding, and visibility of the component. The component developer knows about the implementation details, and sees the component as a white box. The component user, who integrates one or more components to build a complete application, is typically unaware of the component internals and treats it as a black box. Consequently, developers and users of a component have different needs and expectations, and are concerned with different problems.

The component developer implements a component that could be used in several, possibly unpredictable, contexts. Therefore, he has to provide enough information 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 pre-conditions, post-conditions, 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 data-flow 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.

Most of the above information could be computed if the component source code were available. Unfortunately, this is seldom the case when a component is provided by a third party. Typically, the component developer does not want to disclose too many details about his component. The source code is an example of a kind of information that the component developer does not want to provide. Other possible examples are the number of defects found in the previous releases of the component or the algorithmic details of the component functionality. Metadata lets the component developer provide only the information he wants to provide, so that the component user can accomplish the task(s) that she wants to perform without having knowledges about the component that are supposed to be private.

To exploit the presence of metadata, the component user needs a way of knowing what kind of additional information is packaged with a given component and a means of querying for a specific piece of information. The type of

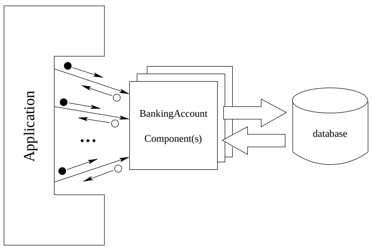
[1] Throughout the remainder of the paper, we use "component user" and "application developer" interchangeably.

information required may vary depending on the specific needs of the component user. She may need to verify that a given component satisfies reliability or safety requirements for the application, to know the impact of the substitution of a component with a newer version of the same component, or to trace a given execution for security purposes. The need for different information in different contexts calls for a generic way of providing and retrieving such information.

Whereas it is obvious that a component user may require the above information, it is less obvious why a component developer would wish to put effort into computing and providing it. From the component developer's point of view, however, the ability to provide this kind of information may make the difference in determining whether the component is or can be selected by a component user who is developing an application, and thus, whether the component is viable as a product. Moreover, in some cases, provision of answers may even be required by standards organizations — for instance, where safety critical software is concerned. In such cases, the motivation for the component developer may be as compelling as the motivation for the component user.

3 Motivating Example

In this section, we introduce the example that will be used in the rest of the paper to motivate the need for metadata and to show a possible use of this kind of information.



(a)

```
public class BankingAccount {
   ...
   BankingAccount(String code) {...}
   ...
   public void open()
             throws CantOpenException,
             InvalidPINException {...}
   public float getBalance() {...}
   public float withdraw(float amount) {...}
   public float deposit(float amount) {...}
   public float moveFunds(BankingAccount destacct,
                          float amount) {...}
};
```

(b)

Fig. 1. (a) High-level view of the application. (b) Interface of the `BankingAccount` component.

The example consists of part of a distributed application for remote banking. The application uses one or more instances of an externally-developed component to access a remote database containing the account-related information. Figure 1(a) provides a high-level view of the application, to show the interaction between the user code and the component(s). Figure 1(b) shows the subset of the `BankingAccount` component interface used by the application. We assume the common situation in which the component user is provided with the interface of the component together with some kind of user documentation.

```
...
public boolean checkingToSavings(String cAccountCode,
                                 String sAccountCode,
                                 float amount) {
1.  BankingAccount checking(cAccountCode);
2.  BankingAccount saving(sAccountCode);
3.  float balance, total;
    ...
4.  checking.open();
5.  saving.open();
    ...
6.  balance = checking.moveFunds(saving, amount);
    ...
7.  total = balance + additionalFunds;
    ...
}
...
```

Fig. 2. Fragment of the application code.

Figure 2 shows a fragment of the application code. The code is part of a method whose semantics is to move a given amount of money from a checking account to a savings account. The first two parameters of the method are two strings containing the codes of the checking account and of the savings account, respectively. The third parameter is a number representing the amount of the funds to be moved. Note that, for the sake of the presentation, we have simplified the example to make it smaller, self contained, and more understandable.

4 Metadata

When integrating an externally-developed component into a system, we may need to perform a set of tasks including, among possible others, the gathering of third-party certification information about the component, analyses and testing of the system, and assessment of some quality of the resulting application. These tasks require more than the mere binary code together with some high level description of the component's features. Unfortunately, the source code for the component is generally unavailable, and so is a formal specification of the component. Moreover, we are not simply interested in having a specific kind of information about the component, as a specification would be, but rather we need a way of providing different kinds of information depending on the context. This is the idea behind the concept of metadata: to define an infrastructure that lets the component developer (respectively, user) add to (respectively, retrieve from) the component the different types of data that are needed in a given context or for a given task. Obviously, metadata can also be produced for internally-developed components, so that all the components that are used to build an application can be handled in an homogeneous way.

This notion of providing metadata with software components is highly related to what electrical engineers do with hardware components: just as a resistor is not useful without its essential characteristic such as resistance value, tolerance, and packaging, so a software component needs to provide some information about itself to be usable in different context. The more metadata that are available from or about a component, the fewer will be the restrictions on tasks that can be performed by the component user, such as applicable program analysis techniques, model checking, or simulation. In this sense, the availability of metadata for a component can be perceived as a "quality mark" by an application developer who is selecting the components to deploy in her system.

Metadata range from finite-state-machine models of the component, to QoS[2]-related information, to plain documentation. In fact, any software engineering artifact can be a metadatum for a given component, as long as (1) the component developer is involved in its production, (2) it is packaged with the component in a standard way, and (3) it is processable by automated development tools and environments (including possibly visual presentation to human users).

As stated in the Introduction, the idea of providing additional data — in the form of either metadata or metamethods returning the metadata — together with a component is not new. The properties associated with a JavaBean [1] component are a form of metadata used to customize the component within an application. The *BeanInfo* object associated with a JavaBean component encapsulates additional kinds of metadata about the component, including the component name, a textual description of its functionality, textual descriptions of its properties, and so on. Analogously, the interface *IUnknown* for a DCOM [3] component permits obtaining information (i.e., metadata) about the component interfaces. Additional examples of metadata and metamethods can be found in other component models and in the literature [14, 20, 8]. Although these solutions to the problem of how to provide additional data about a component are good for the specific issues they address, they lack generality. Metadata are typically used, in existing component models, only to provide generic usage information about a component (e.g., the name of its class, the names of its methods, the types of its methods' parameters) or appearance information about GUI components (e.g., its background and foreground colors, its size, its font if it's a text component). To date, no one has explored metadata as a general mechanism for aiding software engineering tasks, such as analysis and testing, in the presence of components.

To show a possible situation where metadata are needed, let us assume that the component user that we met in Section 3 had to perform two different software engineering tasks on her application: implementation of a run-time checking mechanism and program slicing. We refer to the system in Figures 1 and 2 to illustrate the two tasks.

[2] Quality of Service

4.1 Self-checking Code

Suppose that the component user is concerned with the robustness of the application she is building. One way to make the system robust is to implement a run-time checking mechanism for the application [9, 15]. A run-time check mechanism is responsible for (1) checking the inputs of each call prior to the actual invocation of the corresponding method, (2) checking the outputs of each call after the execution of the corresponding method, and (3) suitably reacting in case of problems.

It is worth noting that these checks are needed even in the presence of an assertion-based mechanism in the externally-developed component. For example, the violation of an assertion could imply the termination of the program, which is a situation that we want to avoid if we are concerned with the robustness of our application. Moreover, according to the design-by-contract paradigm, a client should be responsible for satisfying the method pre-condition prior to the invocation of such method.

```
public class BankingAccount {
  //@ invariant ( ((balance > 0) || (status == OVERDRAWN)) && \
  //@             ((timeout < LIMIT) || (logged == false)) );

  public void open() throws CantOpenException,
                            InvalidPINException {
    //@ pre (true);
    //@ post (logged == true)
  }

  public float getBalance() {
    //@ pre (logged == true);
    //@ post ( ((return == balance ) && (balance >= 0)) || \
    //@         (return == -1.0) );
  }

  public float withdraw(float amount) {
    //@ pre ( (logged == true) && \
    //@        (amount < balance) );
    //@ post ( (return == balance' ) && \\
    //@         (balance' == balance - amount) );
  }

  public float deposit(float amount) {
    //@ pre (logged == true);
    //@ post ( (return == balance' ) && \\
    //@         (balance' == balance + amount) );
  }

  public float moveFunds(BankingAccount destination, float amount) {
    //@ pre ( (logged == true) && \
    //@        ((amount < 1000.0) || (userType == ADMINISTRATOR)) && \
    //@        (amount < balance) );
    //@ post ( (return == balance' ) && \
    //@         (balance' == balance - amount) );
  }
};
```

Fig. 3. Fragment of the component code.

The run-time checks on the inputs and outputs are performed by means of checking code embedded in the application. This code can be automatically generated by a tool starting from a set of pre-conditions, post-conditions, and invariants compliant with a given syntax understood by the tool. As an alternative, the checking code can be written by the application developer starting from the same conditions and invariants. A precise description of the way conditions and invariants can be either automatically used by a tool or manually used by a programmer is beyond the scope of this paper. Also, we do not discuss the possible ways conditions and invariants can be available, either directly provided by the programmer or automatically derived from some specification. The interested reader can refer to References [9] and [15] for details.

The point here is that, if the application developer wants to implement such a mechanism, she needs pre- and post-conditions for each method that has to be checked, together with invariants. This is generally not a problem for the internally-developed code, but is a major issue in the presence of externally-developed components. The checking code for the calls to the external component cannot be produced unless that external component provides the information that is needed. Referring to the example in Figure 1, what we need is for the `BankingAccount` component to provide metadata consisting of an invariant for the component, together with pre- and post-conditions for each interface method.

Figure 3 provides, as an example, a possible set of metadata for the component `BankingAccount`.[3] The availability of these data to the the component user would let her implement the run-time checks described above also for the calls to the externally-developed component. The task would thus be accomplished without any need for either the source code of the component or any other additional information about it.

4.2 Program Slicing

Program slicing is an analysis technique with many applications to software engineering, such as debugging, program understanding, and testing. Given a program P, a program *slice* for P with respect to a variable v and a program point p is the set of statements of P that might affect the value of v at p. The pair $< p, v >$ is known as a *slicing criterion.* A slice with respect to $< p, v >$ is usually evaluated by analyzing the program, starting from v at p, and computing a transitive closure of the data and control dependences.

To compute the transitive closure of the data and control dependences, we use a slicing algorithm that performs a backward traversal of the program along control-flow paths from the slicing criterion [11]. The algorithm first adds the statement in the slicing criterion to the slice and adds the variable in the slicing criterion to the, initially empty, set of *relevant variables.* As the algorithm visits a statement s in the traversal, it adds s to the slice if s may modify (define) the value of one of relevant variables v. The algorithm also adds those variables that

[3] For sake of brevity, when the value of a variable V does not change, we do not show the condition "V' = V" and simply use V as the final value instead.

are used to compute the value of v at s to the set of relevant variables. If the algorithm can determine that s definitely changes v, it can remove v from the relevant variables because no other statement that defines v can affect the value of v at this point in the program. The algorithm continues this traversal until the set of relevant variables is empty.

Referring to Figure 2, suppose that the application developer wants to compute a slice for her application with respect to the slicing criterion $< \texttt{total}, 7 >$.[4] By inspecting statement 7, we can see that both `balance` and `additionalFunds` affect the value of `total` at statement 7. Thus, our traversal searches for statements that may modify `balance` or `additionalFunds` along paths containing no intervening definition of those variables. Because statement 6 defines `balance`, we add statement 6 to the slice. We have no information about whether `checking` uses its state or its parameters to compute the return value of `balance`. Thus, we must assume, for safety, that `checking`, `saving`, and `amount` can affect the return value, and include them in the set of relevant variables. Because `balance` is definitely modified at statement 6, we can remove it from the set of relevant variables. At this point, the slice contains statements 6 and 7, and the relevant variables set contains `amount`, `checking`, and `saving`.

When the traversal processes statement 5, it adds it to the slice but it cannot remove `saving` from the set of relevant variables because it cannot determine whether `saving` is definitely modified. Likewise, when the traversal reaches statement 4, it adds it to the slice but does not remove `checking`. Because the set of relevant variables contains both `checking` and `saving`, statements 1 and 2 are added to the slice and `cAccountCode` and `sAccountCode` are added to the set of relevant variables. When the traversal reaches the entry to `checkingToSavings`, traversal must continue along calls to this method searching for definitions of all parameters. The resulting slice contains all statements in method `checkingToSavings`.

There are several sources of imprecision in the slicing results that could be improved if some metadata had been available with the component. When the traversal reached statement 6 — the first call to the component — it had to assume that the state of `checking` and the parameters to `checking` were used in the computation of the return value, `balance`. However, an inspection of the code for `checking.moveFunds` shows that `saving` does not contribute to the computation of `balance`. Suppose that we had metadata, provided by the component developer, that summarized the dependences among the inputs and outputs of the method.[5] We could then use this information to refine the slicing to remove some of the spurious statements.

Consider again the computation of the slice for slicing criterion $< \texttt{total}, 7 >$, but with metadata for the component. When the traversal reaches statement 6, it

[4] Also assume that the omitted part of the code are irrelevant to the computation of the slide.

[5] We may be able to get this type of information from the interface specifications. However, this kind of information is rarely provided with a component's specifications.

uses the metadata to determine that `saving` does not affect the value of `balance`, and thus does not add `saving` to the set of relevant variables at that point. Because `saving` is not in the set of relevant variables when the traversal reaches statement 5, statement 5 is not added to the slice. Likewise, when the traversal reaches statement 2, statement 2 is not added to the slice. Moreover, because `saving` is not added to the slice, `sAccountCode` is not added to the set of relevant variables. When the traversal is complete, the slice contains only statement 1, 3, 4, 6, and 7 instead of all statements in method `checkingToSavings`. More importantly, when the traversal continues into callers of the method, it will not consider definitions of `sAccountCode`, which could result in many additional statements being omitted from the slice. The result is a more precise slice that could significantly improve the utility of the slice for the application developer's task.

5 Implementation of the metadata framework

In this section, we show a possible implementation of the metadata framework. The proposed implementation provides a generic way of adding information to, and retrieving information from, a component, and is not related to any specific component model. To implement our framework we need to address two separate issues: (1) what format to use for the metadata, and (2) how to attach metadata to the component, so that the component user can query for the kind of metadata available and retrieve them in a convenient way.

5.1 Format of the Metadata

Choosing a specific format suitable for all the possible kind of metadata is difficult. As we stated above, we do not want to constrain metadata in any way. We want to be able to present every possible kind of data — ranging from a textual specification of a functionality to a binary compressed file containing a dependence graph for the component or some kind of type information — in the form of metadata. Therefore, for each kind of metadata, we want to (1) be able to use the most suitable format, and (2) be consistent, so that the user (or the tool) using a specific kind of metadata knows how to handle it.

This is very similar to what happens in the Internet with electronic mail attachment or file downloaded through a browser. This is why we have decided to rely on the same idea behind MIME (Multi-purpose Internet Mail Extensions) types. We define a metadata type as a tag composed of two parts: a type and a subtype, separated by a slash. Just like the MIME type "application/zip" tells, say, 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, as long as we are consistent (i.e., as long as there is a one-to-one relation between the format of the information and the type of the metadatum).

By following this scheme, we can define an open set of types that allows for adding new types and for uniquely identifying the kind of the available data. A metadatum is thus composed of a header, which contains the tag identifying its type and subtype, and of a body containing the actual information. We are currently investigating the use of XML [19] to represent the actual information within a metadatum. By associating a unique DTD (Document Type Definition) to each metadata type, we would be able to provide information about the format of the metadatum body in a standard and general way. We are also investigating a minimum set of types that can be used to perform traditional software engineering tasks, such as testing, analysis, computation of static and dynamic metrics, and debugging.

5.2 Producing and Consuming Metadata

As for the choice of the metadata format, here also we want to provide a generic solution that does not constrain the kinds of metadata that we can handle. In particular, we want to be as flexible as possible with respect to the way a component developer can add metadata to his component and a component user can retrieve this information. This can be accomplished by providing each component with two additional methods: one to query about the kinds of metadata available, and the other to retrieve a specific kind of metadata. The component developer would thus be in charge of implementing (manually or through a tool) these two additional methods in a suitable way. When the component user wants to perform some task involving one or more externally-developed components, she can then determine what kind of additional data she needs, query the components, and retrieve the appropriate metadata if they are available.

Flexibility can benefit from the fact that metadata do not have to be provided in a specific way, but can be generated on-demand, stored locally, stored remotely, depending on their characteristics (e.g., on their amount, on the complexity involved in their evaluation, on possible dependences from the context that prevent summarizing them). As an example, consider the case of a dynamically-downloaded component, provided together with a huge amount of metadata. In such a situation, it is advisable not to distribute the component and the metadata at the same time. The metadata could be either be stored remotely, for the component to retrieve them when requested to, or be evaluated on demand. With the proposed solution, the component developer can choose the way of providing metadata that is most suitable for the kind of metadata that he is adding to the component. The only constraint is the signature of the methods invoked to query metadata information and to retrieve a specific metadatum, which can be easily standardized.

5.3 Metadata for the Example

Referring to the example of Section 3, here we provide some examples of how the above implementation could be developed in the case of an application built using JavaBeans components.

We assume that the `BankingAccount` component contains a set of metadata, among which are pre-conditions, post-conditions, and invariants, and data-dependence information. We also assume that the methods to query the component about the available metadata and to retrieve a given metadatum follows the following syntax:

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

When the application developer acquires the component, she queries the component about the kind of metadata it can provide by invoking the method `BankingAccount.getMetadataTags()`. Because this query is just an invocation of a method that returns a list of the tags of the available metadata, this part of the process can be easily automated and performed by a tool (e.g., an extension of the JavaBeans BeanBox). If the tags of the metadata needed for the tasks to be performed (e.g., `analysis/data-dependency` and `selfcheck/contract`) are in the list, then the component user can retrieve them. She can retrieve the invariant for the component by executing

```
BankingAccount.getMetadata("selfcheck/contract", params),
```

where `params` is an array of strings containing only the string "invariant," and obtain the post-condition for `getBalance` by executing

```
BankingAccount.getMetadata("selfcheck/contract", params),
```

where `params` is an array of strings containing the two strings "post" and "getBalance." Similar examples could be provided for the retrieving of the other information to be used for the tasks.

Our intention here is not to provide all the details of a possible implementation of the framework for a given component model, but rather to give an idea of how the framework could be implemented in different environments, and how most of its use can be automated through suitable tools.

6 Conclusion

In this paper, we have motivated the need for various kinds of metadata about a component that can be exploited by application developers when they use the component in their applications. These metadata can provide information to assist with many software engineering tasks in the context of component-based systems. We focused on testing and analysis of components, and with the help of an example discussed the use of metadata for two tasks that a component user might want to perform on her application: generating self-checking code and program slicing. In the first case, the availability of metadata enabled the task to be performed, whereas in the second case, it improved the accuracy and therefore the usefulness of the task being performed. These are just two examples of the kinds of applications of metadata that we envision for distributed component-based systems.

We have presented a framework that is defined in a general way, so to allow for handling different kinds of metadata in different application domains. The framework is based on (1) the specification of a systematic way of producing and consuming metadata, and (2) the precise definition of format and contents of the different kinds of metadata. This approach will ease the automated generation and use of metadata through tools and enable the use of metadata in different contexts.

Our future work includes the identification and definition of a standard set of metadata for the most common software engineering activities, and an actual implementation of the framework for the JavaBean component model.

Acknowledgments

Gregg Rothermel provided suggestions that helped the writing of the paper. The anonymous reviewers and the workshop discussion supplied helpful comments that improved the presentation. This work was supported in part by NSF under NYI Award CCR-0096321 and ESS Award CCR-9707792 to Ohio State University, by funds from the State of Georgia to Georgia Tech through the Yamacraw Mission, by a grant from Boeing Aerospace Corporation, by the ESPRIT Project TWO (Test & Warning Office - EP n.28940), by the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project. This effort was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061,[6] and by the National Science Foundation under Grant Number CCR-9701973.

References

1. Javabeans documentation. http://java.sun.com/beans/docs/index.html, October 2000.
2. A. W. Brown and K. C. Wallnau. Enginnering of component-based systems. In A. W. Brown, editor, *Component-Based Software Engineering*, pages 7–15. IEEE Press, 1996.
3. N. Brown and C. Kindel. *Distributed Component Object Model protocol: DCOM/1.0.* January 1998.
4. R. Cherinka, C. M. Overstreet, and J. Ricci. Maintaining a COTS integrated solution — Are traditional st atic anallysis techniques sufficient for this new programming methodology? In *Proceedings of the International Conference on Software Maintenance*, pages 160–169, November 1998.
5. J. Cook and J. Dage. Highly reliable ungrading components. In *Proceedings of the 21st International Conference on Software Engine ering*, pages 203–212, May 1999.
6. The common object request broker: Architecture and specification, October 2000.

[6] The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

7. Enterprise javabeans technology. http://java.sun.com/products/ejb/index.html, October 2000.
8. G. C. Hunt. Automatic distributed partitioning of component-based applications. Technical Report TR695, University of Rochester, Computer Science Department, Aug. 1998. Tue, 29 Sep 98 18:13:17 GMT.
9. N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, 1990.
10. T. Lewis. The next $10,000_2$ years, part II. *IEEE Computer*, pages 78–86, May 1996.
11. D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 421–430. IEEE Computer Society Press, 1999.
12. U. Lindquist and E. Jonsson. A map of security risks associated with using cots. *IEEE Computer*, 31(6):pages 60–66, June 1998.
13. D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.
14. G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 1–14. The USENIX Association, 1999.
15. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.
16. C. Szyperski. *Component Oriented Programming*. Addison-Wesley, first edition, 1997.
17. J. Voas. Maintaining component-based systems. *IEEE Software*, 15(4):22–27, July–August 1998.
18. E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September–October 1998.
19. Extensible markup language (xml). http://www.w3.org/XML/, October 2000.
20. Xotcl - extended object tcl. http://nestroy.wi-inf.uni-essen.de/xotcl/, November 2000.