A metadata extracting tool for software components in grid applications

Jeyarajan Thiyagalingam
Vladimir Getov

Harrow School of Computer Science

A Metadata Extracting Tool for Software Components in Grid Applications

Jeyarajan Thiyagalingam and Vladimir Getov
Harrow School of Computer Sciences, University of Westminster, Watford Road,
Northwick Park, Harrow HA1 3TP, U.K.
Email:V.S.Getov@westminster.ac.uk

Abstract

Component-based programming aims at producing higher quality software, increasing the reuse of components and permitting late composition. In the context of component-based programming, applications are treated as composition of components. Given an application composition, some of the components might have been developed outside the context of the application or its domain. As a result, the overall efficiency of the composition, in terms of cost and performance, becomes non-deterministic — may not be guaranteed to be efficient enough, even if the individual components have been proven to be efficient. In other words, two primary goals of software practice, efficiency and quality, do conflict with each other. In this paper, we argue that, this problem can partly be overcome by paying more attention to component-specific information, component metadata, during composition. We describe a possible means of extracting and organising the metadata and formats for specifying the metadata. Our scheme is independent of component- and programming-models and extensible. We see our work as a precursor to a possible runtime scheme, where we intend to facilitate extraction, maintenance and usage of component metadata at runtime.

Keywords *components, composition, performance, context, metadata*

1. Introduction

Software components are independently developed, separately deployable, self-contained units that form a part of a larger software system. Component-based programming, where a software application is built using components, aims at producing higher quality software, increasing the reuse of components and permitting late composition. In this setup, it is valid and often the case, to have compositions of non-native software components, components which are outside the current application domain. For example a generic mathematical component primarily developed for a financial application may be used inside a computational fluid dynamics application. Such a composition of non-domain-specific components challenges the overall efficiency of the composition (throughout this paper, we use the phrase "efficient component composition" to mean a set of collective properties: optimal performance, optimal cost and optimal resource utilisation). This is because, optimality properties of each component do not necessarily mean optimality to be preserved for a composition it constitutes. In other words, these two goals of software practice, componentisation and efficiency of composition, are conflicting between them. Partly, this is due to the fact that components are developed to be separately deployable and little or no attention is paid about the context of use or possible future compositions, which is impractical.

IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)
0-7695-2643-8/06 $20.00 © 2006 IEEE

Efficient component composition aims to seek a balance between these two conflicting goals of software engineering: efficiency and componentisation. In order to do that we need two different types of information, firstly detailed information about components and secondly the current context information. If each component is augmented with additional data, which is called metadata in the literature [7, 3, 4], describing the characteristics and functional behaviours of the component such that they are accessible outside the component's boundaries, then the composition can be adjusted accordingly. However, behaviour of some of the components may be subject to the current context of use, for instance the underlying architecture. This forces us to consider context information during composition, which at some point may become part of the metadata. In this paper, our reference to the term "metadata" includes context information, if not specified explicitly.

If components in a composition or the application developer do have access to the metadata, efficient component composition is about appropriately using, maintaining and staging the metadata and context information of components. However, the key challenge is to consistently maintaining and using the information, independent of component models and programming models. The main idea behind this paper is to decouple this task of maintaining metadata from component and programming models.

The main contributions of this paper are as follows:

1. We formulate a decoupled mechanism for extracting, maintaining and using the metadata from/in components

2. We develop a prototype framework for consistently extracting, maintaining and facilitating the usage of the metadata

3. We illustrate the use of our framework using two different examples.

4. A clear discussion relating previous work to ours

The rest of this paper is organised as follows: Section 2 serves both as a background and a related work section, where we review some existing component/programming models, sources for extracting metadata and compositional information and previous work in this area. In Section 3 we describe the metadata, classifying them in to various levels and we formulate a means for describing the metadata. Section 4 describes the exact information reported at different levels. The overall functionality of the framework, both as a tool and as an API, is discussed in Section 5. The same section briefly discusses the issues relating to extraction, categorisation, consolidation and maintenance of metadata. Section 6 illustrates two different applications of our framework and Section 7 concludes the paper with directions for further research.

2. Related Work and Background

Paul Kelly *et.al.* describe the THEMIS project [7], aiming at designing a programming model and run-time library to support cross-component performance optimisation by permitting explicit manipulation of computation's iteration space at run-time. Their work demonstrated how the metadata and context information can be used to exploit the resource utilisation and exposed a number of possible optimisations across components for a chosen example. In their work, they explicitly assume that each component carries its own metadata as an internal data structure and methods are provided to query those metadata. In addition to this, the overall composition is also maintained as an internal data structure. They envisioned that future components should be built following the same blue-print.

In [5], Edwards describes a reflective metadata wrappers for formally specified components. Their strategy is to inject a wrapper module inside components (at source level) so that the properties of the components can be explored reflectively, based on the supplied formal specification of the component.

Orso *et.al.* describe component metadata for software engineering tasks [9]. They rely on devel-

IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)
0-7695-2643-8/06 $20.00 © 2006 IEEE

oper supplied annotations or metadata to construct different documents, which will eventually assist in different software engineering tasks, by extracting these annotations whenever necessary.

The recent release of Java [6] adds a new language construct called annotation. Annotation is a generic mechanism for associating declarative information (metadata) with program elements. The compiler is then expected to store the metadata in the class files. Later, the VM or other programs can use these metadata for interacting with the program elements or for changing their behaviour. This approach relies on the underlying compiler to pack the metadata for each component and on the virtual machine for providing support in extracting metadata. Although it is an added advantage that more internal details can be exposed by these developer annotations, the mechanism cannot cross the component boundaries.

All these approaches entirely rely on the developer or designer of the component to supply adequate information. However, in a distributed setup, especially in the context of Grid, where the need to support legacy applications is very crucial, it may not be expected that the components are annotated enough. Further, developers or designers may not necessarily supply useful information. In contrary to these approaches, we rely on automatic extraction of metadata from component binaries, where user provided annotations are supported.

Further, deliverables of components have changed from source code to well sophisticated mechanisms such as packages and assemblies. As part of the sophistication, most of the component models support dynamic introspection to limited amount of information. Examples include object browsers in Microsoft's .NET environment [1]. We intend to advance this functionality one step further by enriching the metadata and to use these information in making compositions efficient.

In doing that, we rely on Fractal [4], which is a component model that natively supports composition. Such a support is enabled through composition specification using its own architecture description language (ADL). In cases, where we cannot rely on Fractal-like component model, it is still possible for us to adhere to a composition specification similar to ADL. Although Fractal component-model (and reference implementation of Fractal, such as ProActive [3]) do support composition, they do not explicitly provide any support for metadata manipulation. The Grid Component Model [2], which builds on the Fractal component model and aims at providing component model to support the Grid environment, includes a support for specifying the metadata for components. However, the specification does not specify how to include metadata nor the details of those metadata.

Our work extends the capabilities of these components models by providing a mechanism for metadata extraction, maintenance and manipulation, especially in a distributed, Grid environment.

3. Component Metadata

The exact details of information which can captured from components may vary depending on the underlying component model and programming model. However, we see that the information which should be treated as metadata (and as context information) can be classified into three different levels.

- Higher-level component information. This is essentially configuration information of components, such as dependence metadata, structural composition of components, and interface details of components.
- Internal-level component information. This is a detailed description of internals of components, such as iteration space, storage layouts used, tunable parameters and alike.
- Context information for a component. This is not associated to the previous two and may vary independently of the other two types of information. Information in this category include, platform specific information, possible context sensitive optimisation settings, and alike.

Information extracted from components do not necessarily have clear boundaries, but generally,

IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)
0-7695-2643-8/06 $20.00 © 2006 IEEE

extracted information can be classified in to these levels. As stated earlier, depending on the programming model and component model chosen, some of the information may not be extractable (for any given level) and some of the information may become overlapping (between levels). In addition to this, the availability of the source code for components may provide additional means of information. However, we restrict our attention to component binaries.

The key point here is to have a decoupled mechanism for accessing, specifying and updating associated metadata independent of the component or programming model. We provide a lightweight, extensible framework to manipulate the metadata of components/applications based on their binary - again assuming that source is not available.

3.1. High-Level Component Information

High-Level component information is mostly associated with structural dependencies between components such as interfaces, bindings, attributes, containment relationships of components or inter-component control/data flow.

This information is readily available for certain programming and component models. For instance, the Fractal component model offers ADL (Architecture Description Language) providing higher-level component information and their interactions. However, this is not always possible, especially in cases where components are developed using without any explicit notion of components. In these cases, other mechanisms have to be sought, such as extending the compile-time information or by using reflection mechanisms if available.

3.2. Internal-Level Component Information

Internal-level component information provides detailed description of internals of components such as iteration space, iteration domain, storage layouts used, etc. Some of the component-specific information may be obtained dynamically through reflection. However, in the setting of a real application with large number of classes, it is not realistic to derive all these information at runtime. It is more convenient to compile all these information offline.

3.3. Context Information

Context information is entirely independent of the former two and updated (either manually or automatically) depending on the context of use. These include any platform- or deployment-specific parameters, optimal values for tunable parameters of components and availability information of components. Wherever possible, target function (which could be performance or cost) is expressed as a function of the context information.

4. Organisation and Description of Metadata

If components do not carry their metadata implicitly and if the associated metadata are kept in a decoupled way, there is a danger that the metadata to become outdated or even dangling. This issue is further complicated by the fact that, in a distributed setup, locations of a component and its associated metadata may not be the same. As a result, the organisation of available information is challenging. We assume that this information is maintained consistently by an authoritative server and accessed or cached appropriately. In addition to this, we intend to make the API responsible for guaranteeing that up-to-date information is supplied.

Since the amount of information is overwhelmingly large, we simplify the organisation in a hierarchical, need-to-know basis. Towards this, the metadata is represented at four different levels (named L0 to L3). In this way, the organisation of the metadata becomes simplified, matches the compositional structures and can handle multiple versions. However, this organisation is transparent to the components or applications - meaning that the format is flexible as required. The API we are intending to provide will serve the data, hiding the underlying organisation of the metadata. This way,

IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)
0-7695-2643-8/06 $20.00 © 2006 IEEE

the applications or components are completely decoupled from the content and organisation of the metadata yet they can access the information. The storage of the metadata is carefully chosen to permit multiple versions while restricting duplicates. In addition to these metadata, the framework maintains a registry for all components, organised according to versioning. The query results for a component is an XML based stream — "component summary". The component summary provides basic information about a component (such as its version, authoritative location of latest code, date of last update) and the links to the collections for three levels of information described in Section 3.

The exact form for describing the metadata has to be descriptive enough for components to make use of that information, extensible and should easily be manipulatable. Since the granularity of the information ranges from very low level detail to application level dependencies, various formats may be preferred at various levels of description. For instance, describing the internal aspects of a component, such as iteration space details, uses and defines are easy using a handle to the intermediate format of the component. However, this would be a serious limitation to portability. Therefore, we make extensive use of XML for the description of metadata. Resulting XML files can either be stored in their native form, or could be stored as text objects inside a database.

4.1. Component Summary

As outlined above, the component summary provides the overall connectivity information for a component. In the overall organisation of the metadata, component summary is realized as zeroth level (L0). The component summary for a component provides the following details: component name, link to parent component summary (if applicable), search keywords, versioning information, authoritative location, and 1st,2nd and 3rd level (L1,L2 and L3) information.

Part of the information represented at this level (L0) can be extracted from ADL, if such a description is available.

4.2. High-level Component Information

The high-level component information (information at L1), as discussed in the previous section, intends to provide the following information: component name , link to parent component summary , roles of components, interfaces (provides/expects), bindings, attributes and containment relationships.

4.3. Internal Level Component Information

Internal aspects of a component are of interest to performance optimisation and resides at Level 2 of the metadata hierarchy. For each component, following details are captured/represented: component name, link to parent component summary, loop specific information (such as iteration space, arrays accessed, traversal order of arrays), data structures (such as arrays) and their storage layouts, "uses" and "defines".

4.4. Context Information of a Component

Context information of a component is set to be at the 3rd level of the metadata hierarchy and the exact details to be captured under the context information may vary depending on the component. Generally, the context information may become available progressively at various stages and across runs. In some cases, automatic capturing these context information is possible and therefore the context information may be updated as necessary, particularly across runs. Examples include, run-time performance information (profile information), hot-path information, most occurring data-sizes across runs and other similar information. However, it is not possible to compile a complete list of interesting information to be described as part of the context information. The proposed metadata format covers wide range of interesting information and to facilitate extensibility by incorporating a owner-configurable format-specific placeholder for specifying additional information.

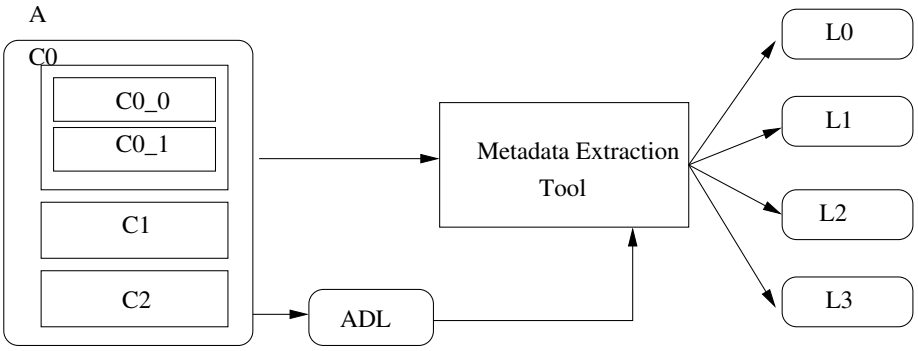

Figure 1. The overall operation of the Metadata extraction tool

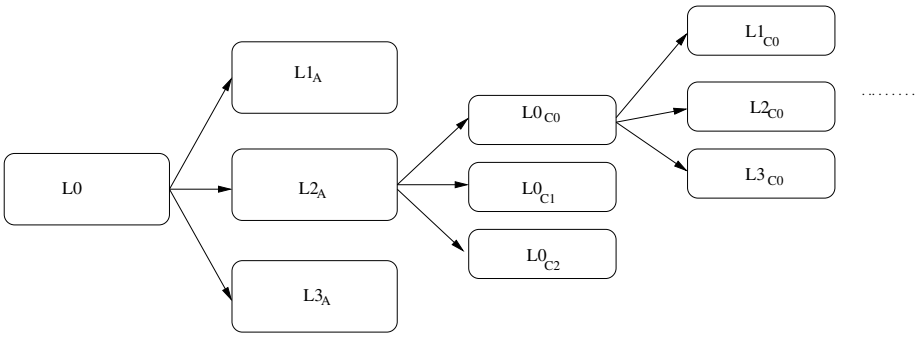

Figure 2. The overall organisation of the extracted metadata

5 Tools and API for Extracting Metadata

To study the feasibility of the idea, we have implemented a prototype tool to extract metadata from components. The tool is implemented in Java, based on the Soot Library [10] and using extensive XML technologies. The extraction process uses the Soot library for manipulating the Java binaries so that all possible information can be extracted from Java binaries. Once extracted, this extracted information is filtered using DTD/XSL technologies for appropriately separating the information into different levels. The tool uses any ADL-based description for complementing the Soot-based extraction process. This process effectively guarantees that component metadata meets the required standards. This process will be recursive given that hierarchical composition exists.

The overall operation of the tool is shown in Figure 1, using an example where an application *A* is produced as a composite of components C_0, C_1 and C_2. The component C_0 is a composite of C_{0_0} and C_{0_1} while the other two components are atomic (no sub-components). The resulting organisation of the metadata is shown in Figure 2. The tool produces different levels of metadata for each of these components. All these information are linked using metadata descriptors at level-0. The tool is still in a prototype stage, but successfully extracts majority of the information associated to the levels 1 and 2 of binary components, written using the ProActive Library. In addition to the metadata at various lev-

IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)
0-7695-2643-8/06 $20.00 © 2006 IEEE

els, the framework maintains a central registry of components along with their version information.

The API part of the tool intends to support querying, updating and maintaining the metadata along with suggestive feedbacks. Currently, metadata are organised and stored hierarchically and mapped to the underlying file system.

6 Applications

The overall goal of our work is to apply the extracted metadata in improving the efficiency of composition of components. Towards this, we illustrate two applications which are under investigation.

6.1 Reconfigurable Software Systems

Reconfiguration of component-based software systems is possible by dynamically managing the composition of components. Such a reconfiguration may be necessary as part of a service provision or as part of a QoS-driven strategies, for instance reconfiguration followed by a failure of a component. One of the crucial features that need to be in place in reconfigurable systems is the capability of finding equivalent components. This is trivial if components are explicitly marked for their equality. However, if such explicit notions are not available, which is possible when introducing new components, the metadata of components can be used to identify equivalent components. However, the task of establishing the equivalency between unknown components is non-trivial. The process involves checking that functional semantics of the concerned components are equal in addition to verifying their interface semantics (inputs, outputs, types etc). This, in turn may have to utilise statement-level metadata. The metadata framework we described in previous sections captures all possible information, including architectural level interfaces and statement-level details. The process of verifying the equality components is currently under investigation.

6.2 Cross Component Performance Optimisation

Optimising the performance of composition is one of the goals of efficient component composition. In [7], Paul Kelly *et.al.* demonstrated the advantages of performing such optimisation assuming that components are well annotated. In our work we intend to extract the metadata from binaries and seeking opportunities for performance optimisation. However, some of the information which are essential for seeking opportunities have already been lost when the extraction takes place. Namely, not all information pertaining to the static analysis are preserved at the binary level. Further, only stage where this performance optimisation can take place is at run-time as no source code is available. Since, we do not have explicit control over the process of JVM jitting, we are forced to consider binary patching when opportunities are detected. We also seeking opportunities to simplify this process.

7 Conclusions and Future Work

In this paper, we have outlined a simple, but comprehensive means for organising the metadata associated with components in a hierarchical fashion. The mechanism we illustrated here in this paper is decoupled from programming and component model and extensible through the API. We then illustrated the application of metadata in performing efficient component composition. We have also highlighted other possible applications of such metadata. However, number of interesting issues remain for addressing:

- In implementing the prototype, we relied on the Fractal component model for simplifying the demonstration process. However, departing from this specific component model may be necessary to demonstrate the applicability of the approach across different component and programming models. In doing that, one of the additional process we might encounter is deriving the compositional information from binaries. Although this could

IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)
0-7695-2643-8/06 $20.00 © 2006 IEEE

be constructed through various methods (e.g. through call graph analysis), the process may prove to be difficult if there are too many compositional patterns or, for instance, if at least one of the components has restrictions in its usage (e.g. obfuscated components).

- Currently, the API or the tool may only facilitate the usage or maintenance of metadata. We would like to investigate how the API or tool could actively provide suggestive feedbacks. Although the tool is in a position to know more about individual components, such a suggestive feedback requires additional knowledge about the domain of the target application/composition. If implemented, such a scheme would improve the overall efficiency of the composition.

- The hierarchical organisation of the metadata may not be optimal if too many nesting levels are present in components. Although this organisation is transparent to the end users, the performance impacts may not necessarily be transparent. More investigation is needed in terms of balancing the management, performance and other issues (such as storage, retrieval and versioning).

- We have not analysed the security issues relating to composition of components. Our assumption that the system level policy would be enforced at composition level may not be valid if domain- or application-specific restrictions need to be placed, perhaps across runs.

- In our framework, extraction of metadata from component binaries is possible given that the binary is Java-byte-code based. As a result, in an application where multi-language components are used, available metadata is restricted to Java-based components. However, in the .NET framework, binaries generated from different source languages share a common intermediate language, MSIL [8], given that source languages are .NET compliant. If we extend our framework to work under the .NET environment, regardless of the source language on which components are based, the framework should be able to extract metadata from all components.

Acknowledgements

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] Microsoft .NET, http://www.microsoft.com/net.

[2] Proposal for a Grid Component Model, CoreGRID Deliverable, D.PM.002, Nov. 2005, accessible via `http://www-sop.inria.fr/oasis/personnel/ludovic.henrio/coregrid/gcm-proposal.pdf`.

[3] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Italy*, volume 2888 of *LNCS*, pages 1226 – 1242. Springer, 2003.

[4] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP2002)*, 2002.

[5] S. H. Edwards. Toward reflective metadata wrappers for formally specified software components, 2001.

[6] D. Flanagan and B. McLaughlin. *Java 1.5 Tiger: A Developer's Notebook*. O' Reilly & Associates, Inc., 2004.

[7] P. H. J. Kelly, O. Beckmann, T. Field, and S. B. Baden. THEMIS: Component Dependence Metadata in Adaptive Parallel Applications. *Parallel Processing Letters*, 11(4):455–470, Dec. 2001.

[8] S. Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[9] A. Orso, M. J. Harrold, and D. S. Rosenblum. Component Metadata for Software Engineering Tasks. In W. Emmerich and S. Tai, editors, *EDO*, volume 1999 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2000.

[10] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.