

Representing Object-Oriented Distributed Systems to Focus the Process of Reverse Engineering

Dan C. Cosma

Radu Marinescu

LOOSE Research Group

“Politehnica” University of Timișoara, Romania

danc@cs.upt.ro, radum@cs.upt.ro

Abstract

When aiming to understand object-oriented distributed applications, reverse engineering processes can model the systems by representing the aspects specific to the object orientation. While these representations provide means for in-depth analysis of various characteristics of the applications, such approaches may ignore their most important traits, specifically those related to the distribution itself. This paper describes the representation of object-oriented distributed systems we have developed to support our reverse engineering approach, and presents the rationale behind the model. The model is built so that the process focuses on the most important aspects of this class of applications, therefore efficiently achieving system understanding by narrowing the search for relevant knowledge. We have applied the process on a couple of case studies, and evaluated the systems by instantiating the model to drive the assessment of the distribution-related characteristics.

Keywords: system analysis and design, software understanding, reverse engineering, distributed systems

1 Introduction

Approaching the goal of understanding software systems through reverse engineering starts by identifying the properties to be studied so that the relevant knowledge is extracted. This is accomplished by defining the actual need of understanding by clearly delimiting the goals, and must continue by expressing them in a way that enables the research to produce a detailed description of the analysis techniques.

As in any scientific or engineering-related field, we need to make use of a representation of the system which, while being only a simplification of the real world, provides all the necessary means for delimiting, measuring, and analyzing the relevant characteristics of the system. The representation must define and describe all the structural or func-

tional units that provide the necessary views on the studied aspects, and to enable easy and precise observations on the attributes of interest.

In itself, a system representation is of little use without considering the related procedure which analyzes the system, the actions that must be done to actually understand the system. Therefore, we believe that any model must be built to address the specific issues related to the process-related goals, and to provide the most relevant views on the software system. The model is an actual part of the methodology of analysis which processes the system entities in an ordered, consistent, and repeatable manner.

This paper describes the reasoning we employed when building a representation for object-oriented distributed systems, and the way this representation addresses the major characteristics of the analyzed software. The model is designed to be used as one of the main components of our methodology of reverse engineering distributed systems we have presented in [3, 2], and is built to focus the analysis process by adhering to the following goals:

- represent the major elements that define the need of understanding a distributed software system, specifically those that are related to the distribution and remote interaction between components
- channel the reverse engineering process on the most relevant system classes, so that the analysis gain important knowledge without applying the most intensive tasks to the entire system

The rest of the paper is organized as follows. Section 2 discusses the distributed nature of the systems, while Section 3 presents the issues we considered when designing the model. Section 4 describes the actual model, and Section 5 presents the application of our model to analyzing several case studies. We discuss in Section 6 the related work, and conclude the paper in Section 7.

2 The distributed aspect

Distributed applications are built for various purposes and needs, but for most of them the distribution-related context actually defines the application's nature, and makes them very different from their 'classic', locally-acting counterparts.

A good example to consider is the case of an electronic mail delivery system. Such applications are widely used, and their utility is beyond argument. Let's assume we look at a mailing system and try to understand its structure without being aware or even considering the possibility that it has a pronounced distribution-related functionality.

If we ignore the distributed nature of the system, we can arrive at a very probable conclusion: the system is very much concerned in storing text (the e-mail) and related binary files (the attachments) in organized databases, and it has advanced features for managing the users that can access and manipulate those files. We will consequently miss the most important functionality the system actually provides, that of sending and receiving from/to remote locations user e-mails.

The main conclusion we can draw from this example is that the understanding of the specific distribution-aware properties of the system is essential for the accurate understanding of the system's actual functionality. Consequently, a representation of a distributed system will have to describe these properties by adopting a perspective that facilitates the valid exploration of the system's distribution and communication-related functionality.

The distribution awareness of a system is best expressed by its interface with the communication medium, by the way it provides or uses resources over the network. To capture the distributed footprint of an application in a representation usable for detailed analysis, we believe that an approach must provide means for describing the following main aspects:

- the set of functionalities (services) the entities in the system provide as available for remote locations
- the remote communication between parties, at a level of abstraction that allows for a good delimitation between the functionalities that belong to the studied system and those provided by external infrastructures
- the relation of between the distribution-related parts of the system, and the ones that only address local concerns

The latter aspect has to be understood not as a clear-cut delimitation between "distributed" and "local" entities, as in most systems such a straightforward approach is hardly realistic or useful for that matter. A representation must

provide a way of describing the degree of involvement in one or the other types of functionality, so that an analysis is able to assess the various roles of the system entities.

3 Building blocks for capturing the distributed nature

Distributed software systems are essentially made of an arbitrary number of processing elements that run at different locations and are interconnected over a network [17].

3.1 Communication Mediator

In most cases, the remote transfer of information is handled by a special communication system, usually external to the application it supports, and often being a fairly complex distributed system itself. Its purpose is to hide the details of transferring data between the remote locations, and to provide higher-level primitives that describe the data formats and control the communication-related activities. It acts as an actual infrastructure the system is built on, and it plays a very important role in both the design of the application, and on its distributed behavior.

To accurately target the analysis, it is important to draw the line that delimits the relevant application-specific system entities from those that only address the communication. By 'relevant' we refer to those system parts that actually represent (and work for) the application's goals, those that implement the design use cases and provide the specific functionality that justified the development of the application.

In our representation of distributed systems we use the term *Communication Mediator* (or, in short, *Mediator*) to designate all the entities, third party or not, that concur to the basic tasks of just sending and receiving data, and are not involved in the actual system-specific functionalities.

As an important component in our model we define the *Application-Mediator Frontier*, as the imaginary line that separates the two sets of entities: on one side will reside only the code fragments belonging to the application, and on the other the ones that implement the communication. Drawing this line is not always easy, and it may be highly related to the task of defining the scope of the analysis.

Nevertheless, there are many cases, where applications rely either on general-purpose communication infrastructures, or on easily separable libraries that deal with the communication details. Examples of widely used infrastructures are CORBA [13], Java/RMI [6], BSD sockets [16], Java JMS providers [11], etc. Considering their relatively limited number and the fact they are widely used, we can safely assume that in these cases the system-Mediator delimitation is easier.

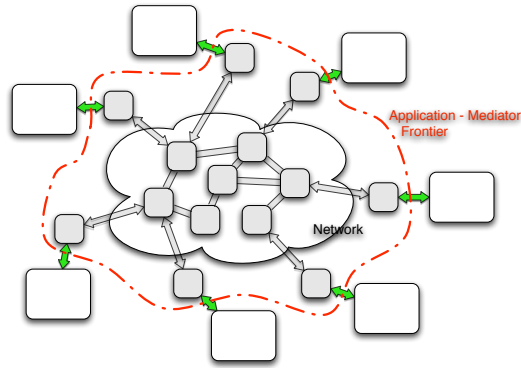


Figure 1. Application-Mediator interaction

Figure 1 shows the relation between the communication mediator and the application components, as a refinement of the previous picture we have drawn. The Mediator entities are the smaller, gray boxes.

3.2 Deployment

When analyzing an application starting from its source code, we have a very particular and somewhat limited view on the system, which remarkably excludes, in many instances, a very important aspect of the distribution: the information describing the actual deployment of the system components over the network. This information is usually scarce, may reside in various places in the system, and it is often difficult to extract.

Therefore, the model must be built so that it follows the realities, rather than limit the approach to the more straightforward cases where the deployment information is readily available.

For this purpose, we make a specifically-targeted, intentional assertion in this matter: *we are certain that we can gather a deep enough knowledge about the system, and can understand it in an accurate and sufficient degree even without using the deployment information at all.* Further more, we theorize that, while definitely useful, *the deployment of the components is far from being the main issue in understanding the functionality of the distributed system.* This assertion is meant to lead us to a methodology of analyzing distributed software that works well using the information that is realistically available in the most frequent industry-specific scenarios, that is, when the only usable artifacts are the programming fragments as encountered in a large, indiscriminating, source code repository.

3.3 Technology dependence

As noted above, distributed applications are usually built over a communication infrastructure (the Mediator) that

deals with all the details of sending and receiving data over the network. The reliance on a Mediator infrastructure imposes a set of rules or constraints the applications must follow in order to use its features. In many cases, these constraints are directly visible in the application's source code, and can be relatively easily detected when needed. For example, BSD sockets imply calling specific library functions, and in Java RMI specific interfaces must be written to describe services, and the classes that implement them can be identified looking at language-specific constructs in the code.

The constraints specific to the technology can prove a valuable tool of determining the relation between the system classes and the distributed aspect, as they are visible in the very parts of the system that communicate over the network.

In our approach, we use these technology-dependent constraints to identify the *frontier classes*, which we define as those classes or interfaces in the system that either directly use Mediator services or they are built to follow Mediator-specific rules in order to export system services or otherwise interface with the Mediator facilities. For example, in RMI, we consider as frontier classes all interfaces that extend `java.rmi.Remote`, and all the classes that call methods of such interfaces.

3.4 Distributable Feature

The representation of a distributed software system that we are building must include the distribution-related entities as explicit elements of the model, so that the analysis to be provided support for their identification. At this point, we make two observations:

- systems usually do not provide a single distributed function or feature, therefore there exist, in fact several groups of classes with important distributed functionality, rather than a single one
- the groups of classes that provide distinct distributed functionalities are not always disjunct entities

To capture the distributed footprint of the system we define an important concept of our model: the *distributable feature*.

A distributable feature is a group of classes that either contribute to a distributed functionality or are themselves users of remote services. The distributable features use or depend on the communication infrastructure to comply to their design goals.

An important observation here is that *the distributed feature is not necessarily a distributed component* deployed at a certain location in the network. A component may consist of several distributable features, and moreover, the

same distributable feature may be provided by several components. In fact, *we aim to detect the relevant distributed characteristics of the system without targeting a thorough component identification*. Instead, we detect distributable features, and analyze the system in respect to them.

3.5 Core entities

Not all the classes in a system have the same importance for characterizing the system. Some classes participate intensely to providing the main features, while some are only accessories to those functionalities.

We believe that a very important aspect for understanding the distributed nature of the system is to accurately focus the analysis on the relevant entities, by isolating a *core set* of classes that represent the distributed functionality. There are two main reasons for this:

- the set may be small in terms of number of classes, therefore easier to understand
- it concentrates the distributed functionality, being highly relevant when trying to understand the distributed nature of the system

Adding to this purpose the concept of distributable features, we can identify within the code, a set of *cores of distributable features*, as smaller sets of highly distribution-aware classes, that represent the main parts of each distributable feature in the system.

3.6 Acquaintance Classes

The classes outside the cores provide more or less distribution-aware functionality, depending on their design. As they are the majority of classes in the system, they cannot be ignored by an analysis. Their relation with the distributed aspect can be analyzed by assessing their relation with the already defined distributable features. We say that system classes have various degrees of *acquaintance* with the distributable features, as a measure of their degree of participation in providing the distributed functionality.

In the model we are building we use the term *feature acquaintance classes*, for designating the classes that do not belong to the cores, and we analyze their characteristics separately.

4 A representation of a distributed software system

Synthesizing the aspects discussed in the previous section, we can define a model that represents distributed software systems, built from a structural-centric point of view, intended to describe the aspects of the system that provide

a good understanding on its distribution-related functionalities.

The model *complements the object-oriented view on the system*, by defining and delimiting the concepts specific to distributed applications.

4.1 Model concepts

To describe a distributed object-oriented software system, the model defines and uses a set of concepts that delimit the relevant entities. They are discussed as follows.

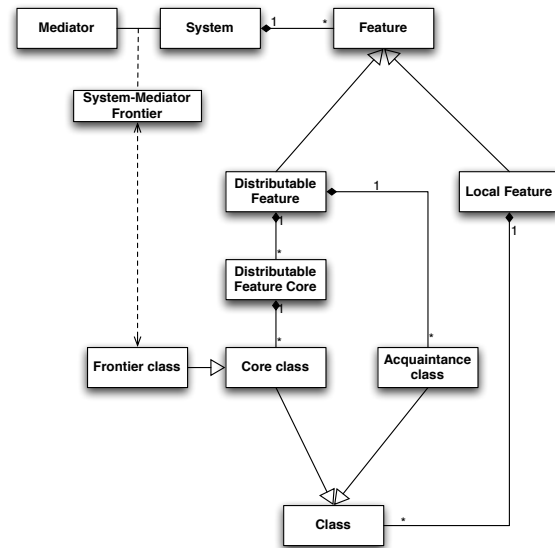


Figure 2. Main model concepts

System. The entity describing the entire software system.

Communication Mediator or, in short, *Mediator*. The infrastructure that provides the means of communicating between remote locations, providing methods for sending, receiving, and otherwise manipulating information over the network. It may consist of operating system services, frameworks, middleware, in-house or 3rd party applications or libraries, and so on. The technology it implies on and the constraints imposed by it must be identifiable and manageable, so that the relation between the system and the Mediator can be characterized in the analysis.

System-Mediator Frontier. The imaginary line that separates the classes belonging to the analyzed system from the entities specific to the communication infrastructure. It is used as a means of identifying and characterizing the particular system classes that directly and consistently act as receivers, senders, subscribers, publishers, etc. of information via the network.

Feature. A *feature* is a part of the system that provides an identifiable functionality in the system. Features can be provided for other system entities to use, or may represent aspects the systems is concerned with when working to fulfill its design goals.

Distributable Feature. A feature to which the distribution-related functionality is central. It may be made of one or more services provided for other features or system entities, or may represent a client functionality for other system or external features. It relies on communicating with remote entities to fulfill its goals, and therefore it contains classes acting at the System-Mediator frontier.

Distributable Feature Core. A minimal subset of the distributable feature classes that concentrate enough distribution-aware functionality so that they can be used to identify the distributable features within the system and characterize their main interactions. All distributable feature cores form the *distributable core* of the system.

Class. A class in the system.

Frontier Class. A system class that directly acts at the frontier with the Communication Mediator by describing, providing or using remote services. It either represents a definition of a service – therefore complying to technology-specific requirements –, or is a class that uses the Mediator to send or receive data over the network, to generate or be informed of remote events, or to otherwise manipulate remote data during the system runtime.

Core Class. A class belonging to a Distributable Feature Core.

Acquaintance Class. A system class that does not belong to the Distributable Feature Core. Its main attribute is the *Feature Acquaintance* which measures the degree in which it participates to one or more of the distributable features in the system. Its involvement decides whether it is actually a part of a distributable feature or is only concerned with local functionalities. Such a class may participate in more than one features, and it can have both distribution-aware and local concerns. This concept models the majority of the classes in a real-world system, where the entities that exclusively provide a single, distribution-related, functionality are relatively few.

Figure 2 shows the concepts discussed in this section, and highlights the relations between them. Figure 3 presents an overview of the model, by exemplifying how a distributed system can be represented in an analysis approach. There are several remarks that must be made at this point:

- The distributable feature *cores* are disjunct entities, having no shared classes between them or with other model entities. While they may be connected one with the other through relations of mutual dependency, these connections must be loose enough to be safe to ignore when analyzing the characteristics of the feature cores in isolation.
- On the other hand, the distributable features themselves *can* share classes, both between them and with the local features provided by the system. In fact, in real-world systems, the shared classes are usually numerous, and the same class can participate in many features at the same time, regardless of their distribution-related characteristic. Consequently, the distinction between the *core* and the larger respective entity becomes a significant one, as the core is able to uniquely identify a feature, and therefore can be utilized with a higher degree of success when characterizing the respective feature and its relations with the rest of the system.
- The local features may include classes that also participate to the distributed aspect. However, they may contain classes entirely separated from the distribution-related functionality. The ratio of predominance of the less involved classes in the system is a very interesting indicator that shows the degree in which the application was actually built for distribution, rather than being mainly focused on local activities.

4.2 Attributes and relations

The model for understanding object-oriented distributed systems can be augmented with a set of attributes and relations respective to the concepts described above. They cover the particular characteristics of the model entities that are considered relevant for understanding the distributed functionality and the structural concerns that need to be identified within the system.

We define the following relations between the entities:

Remote Dependency. Describes the relation established between entities that cooperate over the network (distributable features or distributable feature cores).

Acquaintance. Represents an internal (in-system) dependency established between two entities that work together in a higher or lesser degree. It applies to relations between classes and distributable feature cores, classes and generic class groups, and to the relations between arbitrary groups of classes in the system.

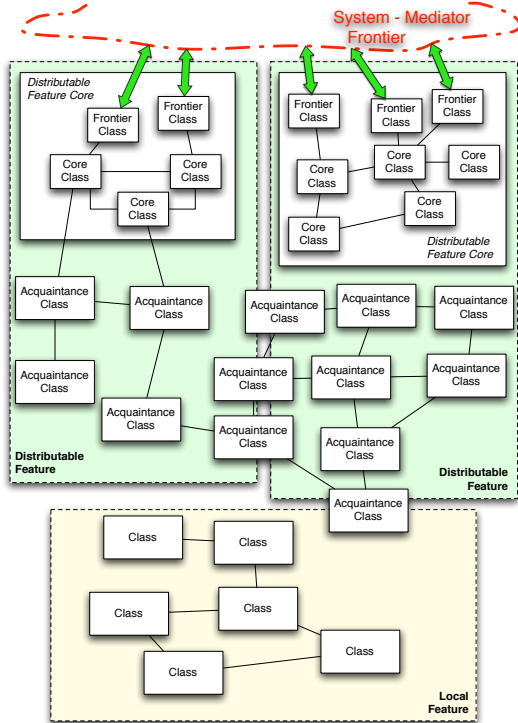


Figure 3. Model overview

For the purpose of characterizing different aspects of the model entities, we use and define a set of generic attributes. During our analysis approach, they translate in numerical values based on software metrics, either calculated directly, or by involving specific algorithms. The attributes are described as follows:

Entity size. Characterizes the proportion of the entity it refers to, so that similar entities can be compared. In our methodology, it is applied to the distributable feature core.

Coupling. Characterizes the strength of dependency between two arbitrary entities of the system. When classes are involved, the attribute refers to the different dimensions of the coupling-related measurements, and it is mainly used to calculate the *degree of acquaintance* (see below).

Degree of acquaintance. This is a calculated attributed, based on *coupling*, meant to characterize the specific relation identified as *acquaintance* above. It is particularly useful to characterize the relations that show the involvement of the acquaintance classes in the various distributable features, or those that show the classes' participation in the provided services.

| Name | NOC | NOC in Distr. Feat. Cores | Distr. Feat. | Average TADF |
|------------|-----|---------------------------|--------------|--------------|
| CAROL | 155 | 28 | 4 | 14 |
| JOTM+CAROL | 218 | 65 | 9 | 29 |
| SPRC | 24 | 9 | 3 | 4 |
| EHCACHE | 93 | 16 | 5 | 9 |
| FWS | 362 | 35 | 2 | 3 |

Table 1. The case-studies in numbers

Distribution awareness. This is an attribute that characterizes the importance the distributed aspect has in the design of the system. Low distribution awareness can suggest a system that wasn't actually or properly built as distributed, and which is mainly a locally acting application that was augmented with a few distribution-aware features with low significance in its functionality.

5 Evaluation of the approach

During the development of our reverse engineering approach, we have analyzed several distributed software projects written in Java, which used RMI for communication (Table 1).

CAROL (<http://carol.objectweb.org/>) and JOTM (<http://jotm.objectweb.org/>) are two medium-sized open-source Java projects. As JOTM was built using the CAROL library we analyzed them together. SPRC is a small student project which allowed us to manually inspect the entire application to evaluate the approach. FWS is a commercial framework for building and executing workflow systems, developed by a local software company, and EHCACHE is a widely used, open-source, Java distributed cache for general purpose caching.

The process analyzes the source code of the application, and extracts, step by step, all the entities we have described as part of the model. The object-oriented view on the system is provided by the *iPlasma* [10] environment, and the tools we have developed are part of this environment.

The extraction starts by identifying the *System-Mediator Frontier* and isolates the *frontier classes*. Next, a core group of classes is gathered around the frontier by a specific algorithm, and then partitioned into a set of *distributable feature cores*. These partitions are considered representatives for the main distribution-aware functionalities of the system, the *distributable features*. As the entire group consists of few classes (relative to the system size, about 10% in the examined systems) they can be analyzed in detail with minimum effort. Table 1 also shows the number of distributable feature cores identified for each analyzed system, as well as the total number of classes included in the cores of each application.

At this point, our methodology provides an overview on the system's distributed architecture, by modeling the *re-*

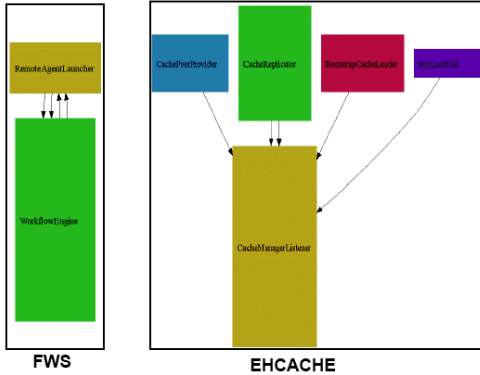


Figure 4. Distributable Feature Cores for two systems

note dependency between the cores. Each communication channel established through the *Mediator* infrastructure is identified, and the relation established between the identified distributable feature representatives is assessed. Figure 4 shows the architectural overview for two case studies.

We must emphasize that we were able to obtain this even though the analysis focused on a relatively small amount of classes (the distributable feature cores). This is a direct consequence of the fact that the model and the underlying process were specifically built to extract the most important trait of the system, its distribution-related characteristics. If we were to analyze the application as a general object-oriented one, we would have been provided with an indiscriminate representation of the system as a set of classes and their OO-specific attributes, rather than with one that facilitated the separation of the relevant sets of classes providing the the important knowledge about the system.

The next steps of the analysis concentrate on the rest of the classes in the system (effectively the 90% of it), and evaluate a single yet relevant attribute for each of them: their *acquaintance* with the distributable feature cores already identified. This provides us with two types of information:

- the overall importance of the distributed aspect in the entire system
- the importance of the distributed aspect for each class

The system-wise attribute provides design-related knowledge about the system, by assessing the relative impact of the distributed and non-distributed functionalities of the system. This can prove useful both for understanding the overall involvement of the system (how “distribution-aware” it really is) and when evaluating the actual state of the software in respect to the original, distribution-related, design goals. The last column in Table 1 shows the importance of distribution in the systems we analyzed, using a

specific metric we have developed for this purpose (the average *Total Acquaintance with Distributable Features*). In short, the metric shows how the system classes are involved with the functionalities represented by the distributable features, the higher the value the higher the impact of distribution.

While involved in calculating the system-level distribution impact, the *acquaintance* attribute also characterizes the classes’ individual involvement in providing distribution-related functionality. This way, our approach is able to detect a series of relevant class-level patterns that delimit three types of involvement: classes that are specifically built for distribution-related tasks, locally-acting classes, and connectors that link these two types of functionality. For example, in the FWS system, this strategy enabled us to directly identify about 80 classes that implemented an entirely non-distributed functionality (a configuration-related tool) which we could consequently easily ignore when aiming to understand the distributed traits of the application. Moreover, we have discovered 5 classes acting as connectors between the two types of functionality (distributed and local) which enabled us to understand the relations between separate subsystems.

6 Related Work

Understanding software systems through reverse engineering is often approach by the researchers. Different approaches use different ways of representing the system, and the relation between the model and the process is valued throughout the approaches. Schmerl et al. [15] address the issue of determining whether an application’s actual architecture is the same as it was originally designed. They develop a framework that facilitates the mapping between the implementation styles to the architectural styles, used at runtime to help detecting patterns that show whether the various actions of the system represent “architecturally significant” activities. Pinzger et al. [12] propose an approach that extracts information from three-tiered distributed software applications based on the COM/COM+ component framework. They build a model that describe the various encountered aspects, such as the persistence, security, transactions and error handling. The abstract model is built using several sources related to the target application. They use the IDL definitions in the source code to understand the descriptions of the components, and search for the particular COM+ statements that indicate architectural characteristics, such as those that are specific to handling transactions. The approach is similar to our case, especially as it is highly aware of the technology-related constraints, by following the major architectural patterns implied by the COM+ framework, and using them to detect the information that provides the system understanding. Han et al. [7]

describe an approach that aims to reconstruct the software architecture for J2EE web applications. To model the system, they use the Siemens Four Views approach [8, 1] to separate the architecture into four views: conceptual, module, execution and code architecture. Li and Tahvildari [9] propose a service-oriented componentization framework for systems written in Java. The purpose is to process an existing Java software system, and transform it into a service-oriented one to support component reuse. The approach models the system as a graph by representing the classes and interfaces as vertices, and labeling them with their name and package information. The edges represent the relations between the classes or interfaces, such as inheritance, realization, aggregation, association, usage, and composition. Eisenbarth et al. [4] propose an approach that aims for the identification of selected features in the source code of a software system. As in our case, they only focus on the features that are considered relevant for the system analyst. The approach uses concept analysis [5] and both static and dynamic analysis to create *feature-unit maps* that describe which system entities implement a set of features considered important. Salah et al. [14] describe an approach for comprehending large software systems based on dynamic analysis. The process extracts different views of the system (use case, module interaction and class interaction) to support the location of features within the application. The analysis is done by performing on the system a set of scenarios extracted from the use cases, and analyzing the execution traces.

7 Conclusions

This paper presented the rationale behind building a system representation designed specifically to support the understanding of object-oriented distributed software. After analyzing the characteristics of the target applications and defining the goals of the approach, we extracted the building blocks that were essential to construct the representation. The model we presented is used in our methodology of reverse engineering distributed systems and facilitates the understanding of the main traits of the software by focusing the analysis on the most important aspects that provide relevant information about the system.

The model uses a small amount of concepts to identify the main entities, and focuses on representing the functional aspects of the system that make it distributed. It was applied on several real-world applications, and the underlying reverse engineering process showed good results in capturing the relevant knowledge, both as a description of the distributed architecture, and the characterization of the impact of distribution.

References

- [1] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford. Documenting software architectures: Views and beyond, 2002.
- [2] D. C. Cosma and R. Marinescu. Distributable features view: Visualizing the structural characteristics of distributed software systems. In *Proceedings of the 4th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*. IEEE CS, 2007.
- [3] D. C. Cosma and R. Marinescu. Understanding the impact of distribution in object-oriented distributed systems using structural program dependencies. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*. IEEE CS, 2008.
- [4] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, vol. 29, pp. 210–224, Mar., 2003.
- [5] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.
- [6] J. Graba. *An Introduction to Network Programming with Java*. Springer-Verlag New York, Inc., 2006.
- [7] M. Han, C. Hofmeister, and R. L. Nord. Reconstructing software architecture for j2ee web applications. In *WCRE '03*, page 67. IEEE CS Press, 2003.
- [8] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [9] S. Li and L. Tahvildari. A service-oriented componentization framework for java software systems. In *WCRE '06*. IEEE CS, 2006.
- [10] C. Marinescu, R. Marinescu, P. Mihancea, D. Rajiu, and R. Wetzel. iPlasma: An Integrated Platform for Quality Assessment of object-oriented design. In *Proc. ICSM '05 (Industrial and Tool Volume)*, 2005.
- [11] R. Monson-Haefel and D. Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.
- [12] M. Pinzger, J. Oberleitner, and H. Gall. Analyzing and understanding architectural characteristics of COM+ components. In *IWPC '03*. IEEE CS Press, 2003.
- [13] A. L. Pope. *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
- [14] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, 00, March 2006.
- [15] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.
- [16] W. R. Stevens, B. Fenner, and A. Rudoff. *UNIX Network Programming*. Prentice-Hall, 2004.
- [17] J. Wu. *Distributed Systems Design*. CRC Press LLC, 1999.