Analyzing the Actual Execution of a Large Software-Intensive System for Determining Dependencies

Trosky B. Callo Arias, Paris Avgeriou
Department of Mathematics and Computing Science
University of Groningen
The Netherlands
trosky@cs.rug.nl, paris@cs.rug.nl

Pierre America
Philips Research and
Embedded Systems Institute
Eindhoven - The Netherlands
pierre.america@philips.com

Abstract

In this paper, we present a dynamic analysis approach to increase the understandability of a large software-intensive system, more particularly to enable the identification of dependencies between its execution entities. This approach analyzes the execution of a software system in a top-down fashion to cope with complexity and uses execution entities such as scenarios, components, and processes rather than code artifacts such as modules, classes, or objects. The approach synchronizes and analyzes two sources of execution information (logging and process activity), and builds architectural views of the system execution, according to a specific metamodel. We have validated this approach on an MRI scanner, a representative large software-intensive system, enabling the identification of dependencies in the execution of its software subsystem.

1. Introduction

Large software-intensive systems often contain millions of lines of code in several different programming languages (heterogonous implementation). Usually, systems of this type have a long history of being exposed to numerous changes. They are typically composed of legacy components associated with large investments and multidisciplinary knowledge spread among the experts of their development organization. To take efficient decisions in the planning and execution of change and maintenance activities, software architects and designers developing this type of systems require up-to-date information that describes the actual system components and their dependencies, e.g. logical, development, physical, process and scenario views of the system [15].

However, due to the complexity of large software-intensive systems, actual information about their components and their dependencies is not always available or accessible, especially information about dependencies within their execution. The development of methods and techniques for dependency analysis is an active research area with considerable attention from the software industry. For instance, many approaches for dependency analysis, analyze execution information based on code artifacts (execution traces), and use techniques such as clustering, filtering, and summarization to manage large amounts of information and eventually present it at higher level of abstraction.

However, the recovery and analysis of execution information of large software systems with heterogeneous implementation, in particular for dependency analysis, is a challenge. This is both a finding of the research community [14], and our own observation as part of our research on evolvability of software-intensive systems [23]. On the one hand, it is hard and expensive to obtain and integrate execution traces from heterogeneous systems. On the other hand, code-base analysis mainly provides information about code artifacts useful to create logical, development, and physical views of a software system. It does not provide enough information about other relevant artifacts within the execution (e.g. data and execution platform), useful to create dynamic views of a system such as process and scenario views. Our focus here is on efficient manners to obtain execution information to identify the various components and the dependencies in the execution of the software of large software-intensive systems.

In this paper, we present a dynamic analysis approach that analyzes and recovers information from the actual execution of a large software-intensive system. To describe the actual execution of a software system, we introduce a metamodel that links high-level abstractions such as scenarios and software components with actual execution activities such as data access, code execution, and platform utilization. This approach synchronizes and analyses two sources of execution in-

formation (logging and process activity) in a top-down fashion. In addition, it builds architectural views of the system execution to enable the identification and description of dependencies between high-level execution entities such as scenarios and software components. We have applied this approach to the software of the industrial large software-intensive system under study in our research, a Magnetic Resonance Imaging (MRI) system developed by Philips Healthcare. The findings of our application and the feedback provided by a group of software architects and designers point that our approach is a structured and problem-driven approach, and it provides actual information to build and maintain process and scenario views of the system.

The organization of the rest of this paper is as follows. In Section 2, we present and discuss related work to describe the motivation of our work. In Section 3, we introduce our approach. Section 4 presents the metamodel used by our approach. Then, we present the specifics of our dynamic analysis in Section 5, describing the sources of execution information and the steps of the analysis. In Section 6, we present the key aspects of the validation of our approach to enable the identification of dependencies in the execution of the MRI software system. Finally, in Section 7 we present some conclusions and future work.

2. Related work

Many methods for the identification of system execution entities and their dependencies are reported in the literature. To motivate our work, we classified related work in two groups, and discuss its applicability to large software-intensive system.

2.1. Code-based methods

The methods in this group analyze code-based information. On the one hand are the methods that statically analyze source code. Moise and Wong reported in [16], that mechanisms of static cross-language analysis still miss information about dynamic behavior between code artifacts. Thus, these methods do not serve our purpose. On the other hand are the methods that analyze execution traces, which in the literature are presented as implementations for specific paradigms. For instance, many tools and techniques are developed to analyze the execution of object-oriented systems [11], e.g. Java [22][6]. Although these methods can be generalized to more languages within the same paradigm, their applicability to large software-intensive systems is limited, mainly because it is not reported how these methods can be integrated to analyze the execution of heterogeneous systems: implemented with different programming languages and different paradigms, and off-the-shelf components [14].

In general, any approach attempting to analyze execution traces of large and heterogeneous software systems will have to address the following two issues. First, techniques to collect execution traces such as source code instrumentation, platform profiling, and compiler profiling may be difficult to apply to heterogeneous and large (millions of lines of code and thousands of files) source code repositories, and to components with partial or no source code available. This is also addressed in [3], but only for pure object oriented implementations. However, the real issue with this techniques is that they are intrusive, which means they create overhead that change the actual execution. Second, it is difficult to obtain high-level information from a large amount of execution traces. This issue is addressed by many approaches in the literature, for instance work for software system analysis presented in [5, 10, 19] proposes summarization and visualization techniques, but it does not cover how to integrate execution traces from a heterogeneous implementation.

2.2. Methods for application management

The methods in this group identify dependencies for application management purposes and analyze monitored information and system repositories [4, 8, 13]. Monitored information represents runtime events such as errors, warnings, and resources usage generated by the system platform (e.g. operating system, middleware, virtual machine). The format and elements within monitored information are generic for all systems running on the platform. System repositories are repositories maintained by the running system platform and contain information related to monitored information and configuration of the system environment. The dependencies in application management are related to the externally observable behavior of system elements [13] such as performance, availability, and other possible end-user-visible metrics [4].

In contrast to methods in the previous group, these methods identify dependencies between major elements of a running system (subsystems, applications, services, data repositories etc.). These major system elements are analyzed as black boxes, which partially facilitates the understanding of the system execution at system level, enables the integration of extracted dependency models into the system documentation, and their reuse in further dependency analysis [4]. The fact that these methods see major system elements as black boxes limits their applicability in a development cycle of large software systems. Usually, unintended or undocumented changes in the implementation elements cause variations of end-user-visible properties, thus in

order to tune these variations a way to zoom in on the implementation elements is required.

Nevertheless, these methods suggest that it is possible to collect actual execution information from sources other than execution traces and system documentation, but it is necessary to bring this information to a level and transparency that software architects and designers can use.

3. The proposed approach

Our approach explores the suggestion we found in the previous work for application management. Its main goal is to help software architects and designers in the identification of system execution entities and their dependencies. We noticed that software architects and designers analyze large software system in a top-down fashion creating mental pictures of what they consider the important parts of the system, which Fowlers define as the architecture [7] . This process allows them to cope with the system complexity and dig down for details when required. Therefore, our approach aims to provide means to collect and analyze high-level information first, and then dig down for details when needed. Figure 1 illustrates the overview of our approach, its inputs, workflow, and output. The input is composed of the execution metamodel (elaborated in Section 4), analysis requirements, and two sources of system execution information (logging and process activity).

The execution metamodel serves two purposes: 1) guides the identification of concepts within the execution of the software system in a top-down fashion, and 2) a blueprint for the creation of execution views. The analysis requirements are relevant to tune the analysis and make the goal of the analysis clear. Previous work in software architecture reconstruction [24] highlights the role of problem elicitation and concept determination activities prior to a reconstruction or analysis process. We do this interacting with experts of the development organization (e.g. managers, architects, and designers). As result of this, we define the scope of the analysis (scenarios), the experts to be involved during the analysis (stakeholders), and ultimately the expected sort of information in the execution views (dependencies) that contributes to solving the specific problem that triggered the analysis. Execution information is collected executing the chosen scenarios with the assistance of some of the identified stakeholders. In particular the system end-users, since in large and complex systems, the intricate functionality can only be captured when it is executed by a qualified end-user. In Section 5, the execution information and the steps of the approach are explained in detail, as well as the role of the experts (software architects and designers) to fine-tune and focus the scope of the analysis.

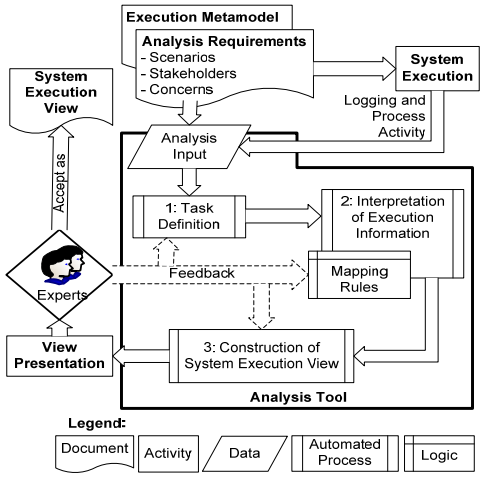

Figure 1. Approach to analyze the execution of a large software-intensive system

4. Metamodel of software execution

We observed that the analysis of the execution of a software system is not a usual activity within its development lifecycle. Often development organizations are not so familiar with abstractions to analyze and understand the execution of a software system as they are with source code artifacts. For instance, a basic high-level description of the execution of a software system is depicted in Figure 2. It describes a top view of the execution of an MRI system in the field, and relates a main execution scenario with its constituent steps, and each step with the required software components that implement it. The usual approach to extend this view will be to map software components to code artifacts. However, code artifacts are not semantically appropriate to describe the execution of a software system, e.g. it is hard to describe and analyze concurrency and communication between software components in terms of classes or methods. Instead, we look at the actual execution and explore the mapping of software components to running processes.

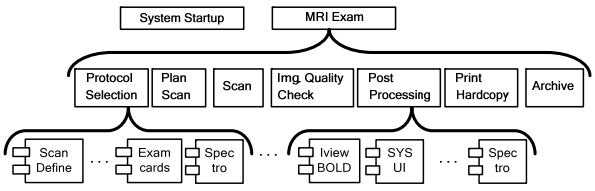

Figure 2. A view of the MRI system execution

The metamodel in Figure 3 illustrates how we take the concepts of execution scenario, task, and software

components as the initial abstractions to describe the execution of a software system. We relate these architectural elements with the generic execution elements (e.g. processes and threads) of the execution platform (operating system) and the type of activities these elements perform in the actual execution of the software system. The metamodel allows illustrating the execution of a software system as a hierarchy that can be navigated from top to bottom. In the rest of this section, we describe the elements in the metamodel and their relationships.

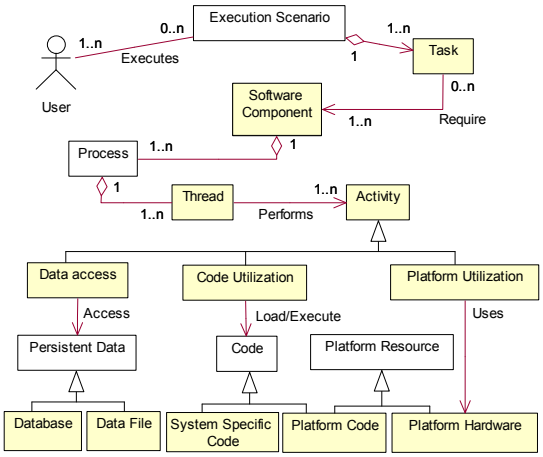

Figure 3. Metamodel of software execution

4.1. Execution scenario, task, and user

At the top, we consider system execution as a set of execution scenarios. A scenario is defined as a brief narrative of expected or anticipated use of a system from both development and end-user viewpoints [12]. Scenarios are also related with use cases, which are frequently used to support the specification of system usage, to facilitate design and analysis, and to verify and test the system functionality. Thus, we describe an execution scenario as the actual execution and usage of the provided system functionality by the intended user.

An execution scenario consists of specific steps, which we call *tasks*, in order to fulfill the intended functionality. Tasks are different from execution scenarios in the degree of complexity and the specialization of their role and functionality. Tasks in an execution scenario implement its workflow. The identification of execution scenarios and tasks corresponds to the stakeholders' interest. Figure 2 shows two execution scenarios at the top, System Startup and MRI Exam. From an end-user (clinical operator) interest, MRI Exam is the main execution scenario, while System Startup may not be too relevant for an end-user. However, in a large software-intensive system, the system startup involves a wide range of interactions and concerns that the development organization wishes to control and analyze, thus it represents a relevant execution scenario. A similar situation applies to the identification of tasks.

4.2. Software components from processes

In our approach, we consider a software component as a set of processes that belong together. A process is an entity handled by the operating system or execution platform hosting the software system. It represents a running application, including its allocated resources: collection of virtual memory space, code, data, and system platform resources. Often in the execution, a large software system is composed of many running applications that interact with each other in order to implement the system functionality. The fact that the development organization identifies these applications as important, reusable, non-context-specific, and often independently deployable units motivate our view of software components. In addition, we describe that one or more running processes identify a software component, because we also take into account actual parent-child relationships between processes (a main processes usually creates other process for a temporary or specific activity), and design relationships that can be used to group different processes.

4.3. Execution activity and resources

A process starts running with a single thread (primary thread), but it can create additional threads from any of its threads. A thread represents code to be executed serially within its process, which at the same time is the realization of execution activity for the utilization of various resources. We classify execution activities in three groups based on the type of the involved resource:

- *Code utilization:* This type of activity represents the execution/loading of code. Code includes executable code from the software component executable file and from (statically or dynamically) loaded libraries (e.g. DLLs). Executables and libraries can be distinguished either as system-specific or as provided by the running platform (platform API). System-specific code elements contain implementation elements (source code libraries or modules) of the software system.
- *Data Access:* This type of activity represent the usage of different sorts of data. A common sort of data is persistent data, which is stored in files and database systems. For instance, data files include configuration parameters, input and output buffers for inter-process communication, and temporary buffers where processes store temporary computations.

- *Platform Utilization:* This type of activity represents the utilization of platform resources. The hosting execution platform makes hardware and software resources available for the executing systems. Hardware resources include CPU time, memory (virtual or physical memory space), and other sorts of hardware devices, that processes access using software resources like APIs and communication services. Executables and libraries provided by the platform are also qualified as platform resources.

5. The dynamic analysis

The dynamic analysis in our approach analyzes the sources of execution information in three steps (see Figure 1): task definition, interpretation of execution information, and the construction of system execution views. An analysis tool supports these three steps. In this section, we describe the sources of execution information and the first two steps. We leave the third step as part of the application of our approach described in Section 6.

5.1. The source of execution information

The source of execution information that we analyze is a combination of logging information and process activity. Large software systems usually store information of their specific activities in dedicated log files, which are already available and well managed by the development organization. Besides debug and test information, logging usually includes information about the workflow of the system functionality (see Figure 4). This information is often controlled and used by testers and specialized users for different purposes [26]. For us, workflow information in logging is an important source of actual information to identify instances of the entities at the top of the metamodel: execution scenarios, tasks, and software components.

Process activity belongs to monitored information (described in Section 2.2), because running platforms offer tools and facilities to monitor or collect process activity. For example, process activity can be monitored with the Process Monitor tool in the Microsoft Windows platform [1], and various open source monitoring tools in the Linux and Unix platforms [25]. Process activity is available in semi-standardized formats independently of the implementation technology and can provide information about activity of non-system-specific entities (e.g. instances of persistent storage, third party software components, platform resources, and even hardware resources). We use process monitoring as a source of information to identify instances of the generic execution elements described at the bottom of the metamodel.

In addition, monitoring process activity and logging are less intrusive as the techniques to collect execution traces (described in Section 2.1). Most platforms provide monitoring facilities, which aim to produce negligible overhead, and in the case of logging, the produced overhead is already part of the normal system behavior. In addition, logging and process activity can simultaneously collect information of the execution of a software system. For instance, Figure 5 shows that for any software system logging (writing) messages in a log file, its monitored process activity, between other events, records the write events in the log file including information such as the writer process. We exploit this fact to synchronize the logging and process activity information in our approach.

Time	*...*	*Subsystem*	*...*	*Description*
...	...	*Scanner*	...	*... Scan Starts*
...	...	*Scanner*	...	*... Executing Acquisition*
...	...	...	...	...
...	...	*Scanner*	...	*… Reconstruction Ends*
…	…	*Scanner*	…	*... Scan Ends*

Figure 4. Example of workflow messages in a log file

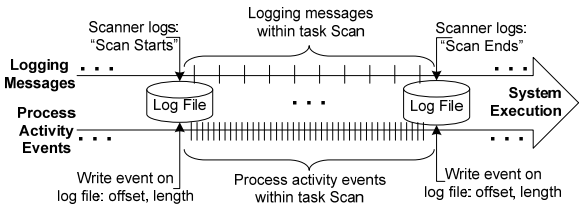

Figure 5. Simultaneous gathering of execution information with logging and process activity

5.2. Task definition

This step is to accomplish with two activities: 1) identification of the tasks that build the workflow of an execution scenario, and 2) synchronization of logging and process activity information for each identified task. Figure 6 shows that the input of this step consists of the log file and the monitored process activity of the executed scenario. It also shows that the output is a set of tasks, where each task is a bundle of sequentially combined logging messages and process activity events, along with a task name and the identification of the software component (SWC) or process that logs the workflow messages (task borders).

Inside task identification, monitored process activity is parsed for write events in the log file. With the information in the write event (offset and length), the logged message is extracted from the log file. A begin-

end text pattern matching indicates if the message is a workflow message that starts or ends a task. In the case it does, the name, logger software component or process, and the borders of the task are identified, both in the log file and in the monitored process activity. Other monitored process activity events and logged messages are added sequentially as execution information of the current identified task. To do this, we assume that 'begin' and 'end' events are logged prior to and right after the task is executed. For situations where our assumption does not hold or the information of the write event (i.e. the offset and length parameters) is not available, the analysis of timestamps is an alternative.

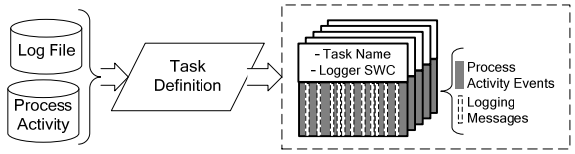

Figure 6. Identifying task information

5.3. Interpretation of execution information

The output from the previous step provides partial information (task and some software components) to describe an execution scenario, but it is still missing information to identify all the actual software components and their interactions. This step sequentially applies a set of mapping rules [24] to extract execution information from our raw data or source views [24], which is the logging messages and process activity events bundled in the tasks of the execution scenario. In our analysis, we define two types of mapping rules:

1) $MR_L(T_{LM})$ = *(subject, verb, object, info)* and

2) $MR_{PA}(T_{PAE})$= *(subject, verb, object, info)*

Where MR_L is for logging information and MR_{PA} is for process activity information. In general, a mapping rule of type MR_L or MR_{PA} takes as argument the text of a logging message (T_{LM}) or the text of a process activity event (T_{PAE}) and generates one or more interaction tuples *(subject, verb, object, info)*. The elements in an interaction tuple identify instances of the elements in the execution metamodel, i.e. *subject:* represents a software component or a process entity, *verb:* represents an execution activity (e.g. read, write, load, and execute), *object*: represents data, code, platform resources, software component and process. Finally, *info:* may contain additional information such as the thread ID. For example, the application of a mapping rule to a process activity event text that describes a running Process B creates another process B1 within its thread T, generates the tuple: *(Process B, Create, Process B1, Thread T)*. Figure 7 illustrates the interpretation process that applies mapping rules (for logging and process activity) to the logging and process activity information of one of the tasks in the execution scenario.

The output is the resulting interaction tuples stored in a graph data structure (interaction graph) to facilitate further analysis and manipulation of the execution information. Additionally in the output is a list of the identified software components and other system resources within the task. Figure 7 also illustrates that the block *Execution Information Interpreter* performs the interpretation step. In our implementation, we use a Python program (including the mapping rules) that parses in a row the data of each task of the scenario. Two groups of Python functions implement the mapping rules. The first group parses the text of logging messages applying some text patterns to extract interactions of architectural entities (e.g. tasks and components). The task definition step also uses part of the group of mapping rules. The second group parses text of process activity events to identify for example: 1) software components structure from process and thread creation events, and 2) data access activity from file I/O events. We will describe some more specific mapping rules in Section 6. Also, in our implementation, the interaction graph data structure is supported by the NetworkX graph library [9], which provide ready-to-use methods to analyze interaction graphs (querying and filtering information) and construct system execution views.

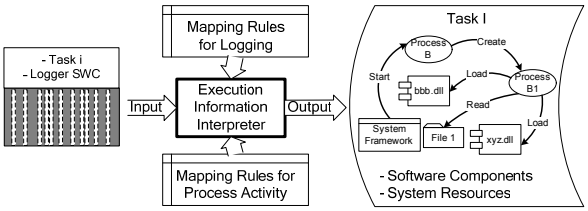

Figure 7. Interpretation of execution information of a task

6. The MRI System as case study

We validated our approach in the context of an actual development project in which our approach enabled the identification of dependencies between execution elements of the Philips MRI software. The MRI system is a large software-intensive system in the healthcare domain. Table 1 summarizes the implementation technology of this system, demonstrating its size and complexity, which makes it a representative case study for our approach. The general condition to identify a dependency between two execution elements A and B, is that A and B interact directly or through a common resource creating a dynamic relationship (e.g. concurrency and communication), which can be described in process or scenario views [15]. Our ap-

proach enabled the identification of this type of dependencies within vertical and horizontal dimensions. First, in the vertical dimension, dependencies relate elements of different types, e.g. the required software components in a task, or the required code and data elements in a scenario. With the information about vertical dependencies, we identify dependencies in the horizontal dimension, which relate elements of the same type. For instance, a dependency between two software components is based on a common data resource used for communication. Similar criteria were used for dependencies between the tasks of a scenario, as well as dependencies between scenarios. In the rest of this section, we describe three key aspects of the application of our approach: the collection of analysis input, the extraction of execution information, and its presentation for dependency analysis.

Table 1. Implementation of the MRI System

Programming Languages	C, C++/STL, C#, Visual Basic, ASP, Jscript, Perl, Batch, and other proprietary languages
Persistent Storage	RDBMS, Flat Files, Indexed and Sequential Files, and XML
Inter-process communication	Socket, COM, shared memory, and file based
Source Code Size	~8 MLOC in ~30 000 files
Involved Disciplines	Physics, mechanics, electronics, medicine, and software

6.1. Collecting the analysis input

Together with a group of software architects and designers, we elaborated the requirements for our analysis: candidate execution scenarios that may be run in the project (along development phases), the involved stakeholders, and hypothetical dependency views (sketches) of the scenarios. The sketched views were mainly guidelines to identify the concerns in the requirements of the analysis. We executed the chosen scenarios with the cooperation of the respective end-user and software designer, who reported no user-noticeable overhead in the system execution. Table 2 summarizes the execution data extracted from the MRI log file, and because the platform of the MRI System is Microsoft Windows, we used the Process Monitor Tool [1] to collect events about file system activity, process activity, and access to the Windows Registry.

Table 2. Execution data for the case study

Scenarios	Logging Messages	Proc. Activity Events
2 in Sys. Testing	~ 31.400	~ 2.5 million
2 in Sys. Verification	~ 24.600	~ 1.1 million
2 in Sys. Release	~ 124.000	~ 2.2 million

6.2. Extracting execution information for dependency analysis

Our analysis tool supports the task definition process providing a user interface. In this interface, the experts can validate the automatic identification of tasks in the chosen scenario and change it if necessary. The validated task definitions (with naming and color code information for each task) are subsequently stored for the next steps in our analysis. Then, we gradually implemented the mapping rules. At first, to extract execution information and populate interaction graphs, then, to identify specific execution elements and their interactions. Table 3 and Table 4 describe the execution elements in the extracted information from logging and process activity data, together with the consulted technical information or documentation to identify text patterns used in the mapping rules for the MRI System.

Table 3. Execution Information from logging

Message Type	Extracted Information	Source of Text Patterns
Workflow	Tasks, Software components, Processes, and Threads	MRI Log Documentation
Debug	Major code modules	

Table 4. Execution Information from P. Activity

Event Type	Extracted Information	Source of Text Patterns
File Access	HW & SW configuration and setting data	MRI naming conventions
	System database data	Filesystem structure and [1]
	DLLs, assemblies, and dynamic wrappers	
	Script programs	
Process activity	Software components, processes, and threads	[1, 18]
Windows Registry access	COM elements	[17, 18]
	HW & SW configuration and setting data	
	Communication services and platform resources	

On the one hand, the elements in the extracted information are specializations of the elements in the metamodel. For instance, to analyze communication dependencies based on the information about communication in Table 1, Figure 9 illustrates the specialization of some platform resources in communication services, system-specific code in COM elements, and some data files in configuration and data repository. Our mapping rules group and classify the extracted

elements to make specific specializations explicit. For instance, often the filesystem structure provides text patterns about the system structure, embedded in the names of files and directories, which we used to classify some files e.g. as data configuration. Some sophisticated text patterns to identify COM classes and interfaces are found in the events accessing the Windows Registry. Similarly, debug messages in the MRI log file provide patterns to identify the call of major code modules. In addition, while digging down in the analysis, the identification of a high-level concept or a specific concern may request the redefinition of some execution entities and mapping rules. For instance when the stakeholders want to analyze the configuration data of a specific hardware device in isolation from the global system configuration, a new mapping rule will be implemented to specialize (group or classify) some of the configuration repository files. On the other hand we consider that only patterns based on logging and naming conventions should change in the extraction process (applied in the MRI) to be considered generic for other large systems running on the same platform.

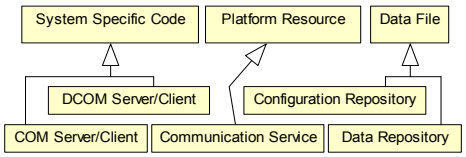

Figure 9. Specialization of execution elements to analyze communication dependencies

6.3. Presentation of execution information for dependency analysis

In our approach we use scenario views to analyze dependencies and in principal to support the top-down strategy. We create scenario views analyzing information in the interaction graphs of a scenario, e.g. common elements, directed paths, and node degrees [9]. We provide two types of scenario views to present execution information in two steps. First graph-based views present overviews of the execution. Second, matrix-based views present details of the execution scenario to zoom into the analysis. In addition our current work aims to build sequence diagrams to provide detail information in the time dimension. Graphs and matrices are usual representations to describe and analyze complex systems. Further details about it and the correspondence between graphs and matrices are presented in [20, 21].

Figure 10 illustrates how our scenario views (an overview and two matrices) present execution information for dependency analysis. The graph overview (a) describes relationships between high-level elements. At the left side of the overview is a stack of the identified tasks in the scenario. Color-coded edges relate a task with one or more software components within the task. The color code identifies the trajectory of a task through the rest of elements in the overview. A record figure represents a software component, where the component name is linked to its set of running processes (fields of the record). The color-coded edges continue and describe interactions at a high level: a software component acts as the subject performing activities on objects; the objects represent identified specializations of data, code, or platform resources. In this case, the overview shows the configuration repository of the system and one of its specializations (UI Menu Structure). Currently our analysis tool use the Dot format and the Dot tool [2] to generate graph overviews.

Table 5 describes the matrix views used in our approach to present and analyze execution information in detail to identify dependencies. Our analysis tool provides filtering facilities to choose the type of elements in the rows, columns, and cells. Matrices I and II are often used to analyze horizontal dependencies. For instance, in Figure 10 a matrix view I (b) is used to identify UI Menu Structure elements commonly accessed in the execution scenario by two of the MRI system components. A matrix view III (c) is used to the Configuration Repository element that the SCANNER component requires (read) within all the tasks in the execution scenario.

Table 5. Matrix views of execution information

	Rows	Columns	Cells
I	Software Components	Software Components	Data, code, and platform elements
II	Tasks	Tasks	
III	Software Components	Tasks	Component's interactions
IV	Software Components	Tasks	Component's processes

6.4. Main Findings

The application of our analysis showed three main findings. First, the top-down approach actually provided high-level dependency views that on the one hand refreshed and validated the mental pictures used by various experts, and on the other hand helped to convey the understanding of the system to other designers. For instance, our experts were able to spot out desired and undesired communication paths using graph overview; recover many implicit concepts ruling the actual utilization (distribution and combination) of configuration data and major code artifacts within the analyzed scenarios.

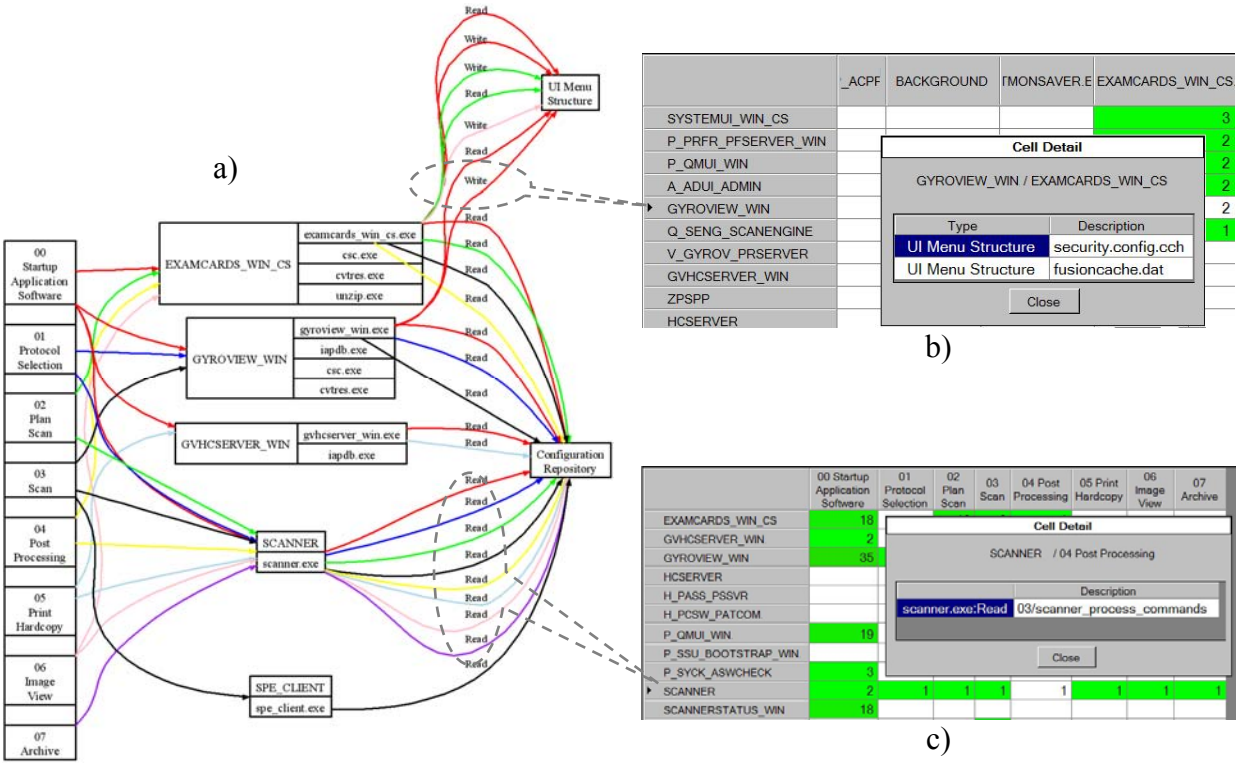

Figure 10. Scenario views with execution information for dependency analysis

Second, many actual aspects of the realization of the implementation were uncovered. For instance, describing the identified software components with our notation (the records in Figure 10 (a)), enabled the quick identification of characteristics such as the implementation technology and the usage of third party or platform utilities to provide the specific characteristic of the system functionality. These two first findings contributed to understand dynamic changes in the software system configuration, and identify guidelines for future reorganization of the configuration data.

Third, we observed the importance of expressing mapping rules in terms of the elements of our metamodel. This made the transformation of implicit information (from logging and process activity) into explicit architectural information transparent and understandable. We can also say that architects and designers did not find it difficult to understand the rationale we used to draw boxes and lines.

7. Conclusions and Future Work

We consider that the main contribution of our approach is centered around two points. First, it is a structured and problem-driven approach. Our software architects and designers appreciated the usage of the metamodel as guideline for the preparation of the analysis and the presentation of results in a top-down fashion. This allows us to tune the analysis, deal with complexity, and generate the required information (views) to address a concrete problem. Second, our software architects and designers got actual information about execution dependencies at the architectural level without being overwhelmed by the complexity of the software system. In addition, because logging and monitoring process activity are non-intrusive; we collected up-to-date execution information without touching the source code or creating overhead.

We believe that the combination of logging and process activity is a valid source of information to analyze the execution of a large software-intensive system, especially for heterogeneous implementations. However, it is necessary to have a structured strategy to exploit this combination. In our case, the metamodel, our mapping rules, and our views were relevant to recover important execution information in a top-down fashion. These elements make our approach extensible, scalable, and transparent. In our future work, we aim to extend, formalize and generalize our metamodel and our current set of mapping rules to analyze the execution of large software-intensive systems in a larger context. Finally, we consider that our approach is complementary to the existing dependency analysis methods, therefore in future work we aim to provide the means to link our approach with the existing methods.

This will allow to the development organization to have complete and actual information about dependencies in its software system.

Acknowledgment We would like to thank the Software Architecture Team and the software designers of the MRI system in Philips Healthcare, as well as Frank Wartena, Rob van Ommering, Richard Doornbos, and our Darwin colleagues for their feedback and joint work. This work has been carried out as a part of the Darwin project at Philips Healthcare under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

References

[1] Process monitor, `http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx`

[2] Graphviz - graph visualization software, `http://www.graphviz.org/`

[3] L. C. Briand, Y. Labiche, and Y. Miao, Towards the reverse engineering of uml sequence diagrams, presented at *10th Working Conference on Reverse Engineering*, 2003.

[4] A. Brown, G. Kar, and A. Keller, An active approach to characterizing dynamic dependencies for problem determination in a distributed environment, presented at *IEEE/IFIP International Symposium on Integrated Network Management*, 2001.

[5] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. v. Deursen, Understanding execution traces using massive sequence and circular bundle views, in *Proceedings of the 15th IEEE International Conference on Program Comprehension*: IEEE Computer Society, 2007.

[6] A. Egyed, A scenario-driven approach to trace dependency analysis, *IEEE Transactions on Software Engineering*, vol. 29, pp. 116-132, 2003.

[7] M. Fowler, Who needs an architect? *IEEE Software*, vol. 20, pp. 11-13, 2003.

[8] M. Gupta, A. Neogi, M. K. Agarwal, and G. Kar, Discovering dynamic dependencies in enterprise environments for problem determination, presented at *14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, 2003.

[9] Networkx, `https://networkx.lanl.gov/`

[10] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, Recovering behavioral design models from execution traces, in *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*: IEEE Computer Society, 2005.

[11] A. Hamou-Lhadj and T. C. Lethbridge, A survey of trace exploration tools and techniques, in *Conference of the Centre for Advanced Studies on Collaborative research*: IBM Press, 2004.

[12] R. Kazman, G. Abowd, L. Bass, and P. Clements, Scenario-based analysis of software architecture, *IEEE Software*, vol. 13, pp. 47--55, 1996.

[13] A. Keller and G. Kar, Dynamic dependencies in application service management, presented at *International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA*, 2000.

[14] K. Kontogiannis, P. Linos, and K. Wong, Comprehension and maintenance of large-scale multi-language software applications, in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*: IEEE Computer Society, 2006.

[15] P. Kruchten, The 4+1 view model of architecture, *IEEE Software*, vol. 12, pp. 42-50, 1995.

[16] D. L. Moise and K. Wong, Extracting and representing cross-language dependencies in diverse software systems, in *Proceedings of the 12th Working Conference on Reverse Engineering*: IEEE Computer Society, 2005.

[17] D. Mar-Elia, How the registry is architected, `http://www.windowsitlibrary.com/Content/224/toc.html`, 2007

[18] MSDN, Processes and threads, `http://msdn2.microsoft.com/en-us/library/ms681917.aspx`, November 2007

[19] H. Safyallah and K. Sartipi, Dynamic analysis of software systems using execution pattern mining, in *Proceedings of the 14th IEEE International Conference on Program Comprehension*: IEEE Computer Society, 2006.

[20] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, Using dependency models to manage complex software architecture, presented at *20th annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2005.

[21] D. Sharman M. and A. Yassine A., Characterizing complex product architectures: Regular paper, *System Engineering*, vol. 7, pp. 35-60, 2004.

[22] T. Systa, Static and dynamic reverse engineering techniques for java software systems, *PhD Thesis*, University of Tampere, 2000

[23] P. van de Laar, P. America, R. Rutgers, S. van Loo, G. Muller, T. Punter, and D. Watts, The darwin project: Evolvability of software-intensive systems, presented at *Third International IEEE Workshop on Software Evolvability* 2007.

[24] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, Symphony: View-driven software architecture reconstruction, presented at *IEEE/IFIP Working Conference on Software Architecture (WICSA'04)*, 2004.

[25] A. Vasudevan, Process monitor how-to for linux, `http://www.linuxdocs.org/HOWTOs/Process-Monitor-HOWTO.html`, 2007

[26] D. J. Yantzi and J. H. Andrews, Industrial evaluation of a log file analysis methodology, in *Proceedings of the 5th International Workshop on Dynamic Analysis*: IEEE Computer Society, 2007.