A Top-Down Approach to Construct Execution Views of a Large Software-Intensive System

<table>
<thead>
<tr>
<th>Journal:</th>
<th>Journal of Software Maintenance and Evolution: Research and Practice, incorporating Software Process: Improvement and Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuscript ID:</td>
<td>Draft</td>
</tr>
<tr>
<td>Wiley - Manuscript type:</td>
<td>Research Article</td>
</tr>
<tr>
<td>Date Submitted by the Author:</td>
<td></td>
</tr>
<tr>
<td>Complete List of Authors:</td>
<td>Callo Arias, Trosky; University of Groningen, Mathematics and Computing Science America, Pierre; Philips Research; Embedded Systems Institute Avgeriou, Paris; University of Groningen, Mathematics and Computing Science</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Reverse architecture, dynamic analysis, software-intensive systems</td>
</tr>
</tbody>
</table>

http://mc.manuscriptcentral.com/jsme
A Top-Down Approach to Construct Execution Views of a Large Software-Intensive System

Trosky B. Callo Arias, Pierre America, and Paris Avgeriou

1 Department of Mathematics and Computing Science - University of Groningen
Groningen – The Netherlands
2 Philips Research and Embedded Systems Institute
Eindhoven - The Netherlands

SUMMARY
This paper presents a top-down approach to construct execution views of a large and complex software-intensive system. Execution views describe what the software does at runtime and how it does it. The presented approach represents a reverse architecting solution that follows a set of pre-defined viewpoints and a metamodel, which capture the runtime perspective on a system of a particular development organization. It applies a dynamic analysis technique to extract runtime information from a combination of system logging and runtime measurements in a top-down fashion. The approach was developed and validated constructing execution views for an MRI system, thus it represents a solution that can be applied on similar large and complex software-intensive systems.

1. INTRODUCTION

Large software-intensive systems development combines various hardware and software elements, which are typically associated with large investments and multidisciplinary knowledge. A particular characteristic of the development of this type of systems is that their software elements take a considerable fraction of the development effort. These software elements contain millions of lines of code, written in several different programming languages (heterogeneous implementation), and influence the design, construction, deployment, and evolution of the system as a whole. These characteristics have lead to a demand for architectural descriptions that can help software architects and designers to get a global understanding of a software-intensive system. Architectural descriptions are organized into one or more architectural views, which are expressions of the system’s architecture using a particular architectural viewpoint [1]. Each architectural view (or simply, view) is usually comprised of one or more models that address some of the architecture-related concerns held by the stakeholders of the system. For example, a devel-

* Correspondence to: Trosky B. Callo Arias. SEARCH–IWI, University of Groningen, P.O. Box 407, 9700AK Groningen, The Netherlands
† E-mail: trosky@cs.rug.nl

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.
development view describes information that addresses concerns related to the organization, reuse, and portability of the software modules that comprise a software system [2].

An important aspect for the use of architectural descriptions is that they should contain up-to-date information about the system at hand, i.e., the actual system elements and the relationships between them. However, due to the complexity of large software-intensive systems, architectural views of this type of systems are not always up-to-date, available or accessible. This is especially the case for systems that have a long history of being exposed to numerous changes and contain legacy components associated with multidisciplinary knowledge spread across the practitioners of their development organizations. Therefore, as part of our research project on evolvability of software-intensive systems [3], our goal is to find ways to construct up-to-date views that practitioners can use to support the development and maintenance of large and complex software-intensive systems.

Our particular focus is the construction of execution views, which we define as views that describe what the software of a software-intensive system does at runtime and how it does it [4]. The term runtime refers to the actual time that the software system is functioning during test or in the field. Execution views are important assets that help practitioners to describe, analyze, communicate, and address the different system stakeholders’ concerns with respect to the runtime structure and behavior of a software system. In contrast to development or logical views, which describe a system in terms source code entities (e.g., modules, classes, functions, or methods), execution views describe a system in terms of high-level runtime concepts (e.g., scenarios, components), actual runtime platform entities (e.g., processes, threads), hardware and data system resources, and the corresponding runtime interactions between them. Therefore, one can get a global understanding of the runtime elements of a system and the relationships between them without being overwhelmed by the size and complexity of the system implementation.

Various techniques are available in the literature to construct views that describe the runtime of a software system. Most of them are reverse engineering techniques that use runtime data based on execution traces of source code entities i.e. functions, methods, classes, and packages. However, applying these techniques on a large and complex software-intensive system with heterogeneous implementations is a major challenge [5]. On the one hand, it is hard to obtain code-based runtime information from heterogeneous and large implementations. On the other hand, views constructed with code-based runtime information do not include instances of actual runtime entities, such as data repositories and hardware entities, which play an important role within the actual runtime structure and behavior of software-intensive systems.

In this paper, we present an approach to construct execution views for a large and complex software-intensive system. The major contribution of this approach is that it enables the top-down construction of execution views. By top-down, we refer to the construction of views with high-level information at first and then, if the stakeholders need it, views with detailed information to dig down in details. This approach follows a set of pre-defined execution viewpoints and a metamodel, which capture the set of concerns of a particular development organization, and the elements and relationships that can be used to describe the runtime of its particular system [4]. The approach is iterative and problem-driven; it involves and collects the feedback of stakeholders (practitioners). It implements a dynamic analysis technique to extract information from runtime data composed by a system log files and runtime measurements.

The approach was developed and validated on a Magnetic Resonance Imaging (MRI) system. This is an industrial large software-intensive system developed by our industrial partner, Philips.
Healthcare [6]. The development and validation have focused on the construction of various execution views of the MRI system [7, 8]. To keep the size of the article manageable, we illustrate the construction of an execution profile view of the MRI system. This type of execution view provides tangible evidence and insights into the runtime structure and behavior of key execution scenarios of the MRI system. The involved practitioners of our industrial partner reported that the constructed view helped them to identify the various system components and their dependencies in a top-down fashion through the planning and downstream execution of development activities related to the involved key scenarios. Through the validation we identified that the approach enables the construction of useful execution views in practice, and it can be considered as a reverse architecting approach [9, 10] for similar large and complex software-intensive systems.

The organization of the rest of this paper is as follows. In Section 2, we present and discuss related work to motivate our approach. In Section 3, we introduce our approach. Section 4 introduces the set of execution viewpoints and the metamodel used by the approach. Then, we present the dynamic analysis technique in Section 5, including the source of execution information. In Section 6, we describe in detail the construction of an execution profile view and its value to illustrate the validation of the approach. In Section 7, we discuss the applicability of our approach. Finally, in Section 8 we present some conclusions and future work.

Note for the editor and reviewers:

This paper is an extension of our previous work presented in [8], which focuses in the presentation of the dynamic analysis technique to identify the runtime components of a large software-intensive system and dependencies between them. The extension presented in this paper reports further development and validation of our research, a reverse architecting solution to construct execution views based on the dynamic analysis technique presented in [8]. A previous version of this paper was not accepted by the same venue. The observations from the reviewers (see Annex A) highlighted that a required 30% delta was not achieved, and the presentation and structure of the validation was weak. In this paper, we have addressed these observations including:

- An overview of the execution viewpoints and the extended metamodel in Section 4.
- An extended description of the dynamic analysis technique including the sources of runtime information and the rule-based technique in Section 5.
- The validation of the approach as the construction of an execution profile view for the MRI system in Section 6.
- A discussion of the applicability and limitations of the approach in Section 7.

2. RELATED WORK

Constructing execution views and other type of architectural views of an existing software system is an architecture reconstruction activity. Architecture reconstruction is defined as the form of reverse engineering in which architectural information is reconstructed for an existing system through a structured procedure applying an abstraction technique and involving the various stakeholders [10]. Most of the work in the literature is about reverse engineering techniques (in the remainder of the section referred to as techniques) that can be used as part of an architecture reconstruction solution. In this section we describe some of the techniques that can be used to reconstruct views that describe the runtime of a software system, e.g., for the identification of system elements and their dependencies. The techniques are presented in three groups distin-
guishing the type of constructed view (module, component-connector, and management), the source of information, and some characteristics that motivate the development of our approach.

2.1. Techniques for dynamic module views

Dynamic module views describe the runtime interactions between code artifacts such as modules, classes, and objects. These views are mainly constructed applying techniques that analyze execution traces, which track code level events at runtime, e.g., the entry and exit of functions and methods. Execution traces are collected using instrumentation techniques such as debugging, compiler profiling, or source code instrumentation [11]. In the literature, these techniques are presented for various programming languages and paradigms, e.g., some tools and techniques are developed to analyze the execution of object-oriented systems [11, 12], such as Java [13, 14]. Although these techniques can be generalized to include more languages within the same programming paradigm, their applicability to construct dynamic module views of software-intensive systems is limited. This is the case when the system implementation is large and heterogeneous, including more than one paradigm and off-the-shelf components with partial or no source code available. To cope with large amounts of execution traces, some of these techniques propose aggregation, summarization, and visualization solutions to abstract execution traces and present them as high-level information [15-17]. However, this high-level information only describes a system in terms of code or implementation entities, missing runtime relations among heterogeneous elements and other actual runtime elements (e.g., hardware and data) and their relations with the described software modules. Therefore, we consider that only looking at or analyzing the source code to describe the runtime of a software-intensive system is not enough and other sources of information need to be explored.

2.2. Techniques for component-connector views

Component-connector views [18] are specially constructed to describe relationships between architectural elements such as components, connectors, interfaces, ports, which are considered more architecturally significant than source code modules. The Component Based Dependency Model (CBDM) [19], Dependency Chains [20] and the Architectural Dependency Graph [21] are examples of techniques that analyze system descriptions or documentation (e.g., design models and architectural descriptions using architectural description languages) to construct component-connector views.

The abstraction level, the type of elements, and the type of relationships described in these views make them suitable to analyze large software systems in a top-down fashion. However, the application of techniques that use system documentation need to overcome with important practical issues within the development of large and complex software-intensive systems [22]. In this context, system documentation is often informal, extensive, and it is not (yet) possible to assure that the included design models or descriptions reflect the actual system realization due to architecture or design erosion and intensive evolution. Finally, extensive system documentation often turns out inaccessible due to complexity of the development organization (e.g., distributed locations and multidisciplinary knowledge) and the need of domain knowledge to understand and process it. Therefore, we consider that to construct up-to-date views of the runtime of a software-
intensive system, it is necessary to find a source of actual up-to-date runtime information, but that can still provide information at the level of system descriptions or documentation.

2.3. Techniques for management views

Management views describe information related to the externally observable behavior of system elements [23] such as performance, availability, and other end-user-visible metrics [24]. Management views of software systems can be constructed using techniques that analyze monitored information and system repositories [23-25]. Monitored information represents runtime events such as errors, warnings, and resources usage generated by the system runtime platform (e.g. operating system, middleware, virtual machine). The format and elements within monitored information are generic for all systems running on the same runtime platform. System repositories are maintained by the runtime platform and contain information related to monitored information and configuration of the system environment.

In contrast to the views constructed by the techniques in the previous groups, management views describe the major elements of a running system (subsystems, applications, services, data repositories etc.) as black boxes. This enables the understanding of the runtime of a system at a system level, the integration of constructed views into the system documentation, and their reuse in further dependency analysis [24]. However, we consider that describing and analyzing major system elements as black boxes limits the application of management views within a development cycle of a large software-intensive system. 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 look into details (e.g. the implementation elements) is required. Nevertheless, these techniques show that it is possible to collect up-to-date runtime information from sources other than execution traces and system documentation. However, it is necessary to bring this information to the level of abstraction and transparency that software architects and designers need to support development and maintenance activities.

3. OVERVIEW OF THE PROPOSED APPROACH

As we described in the related work section (Section 2), the development of our approach was inspired and motivated by the characteristics of existing techniques to construct views that describe the runtime of a software system. Another important factor that inspired the development of our approach is the need of support for top-down analysis that practitioners (mainly architects and designers) conduct to plan and execute development and maintenance activities. Top-down analysis at the architecture level is very similar to top-down program understanding [26]. We observed that in practice architects and designers start top-down analysis using simple diagrams and mental pictures of what they consider the important parts of the system, (which Fowler [27] defines as the architecture). These diagrams and mental pictures are domain models that capture high-level knowledge about the system functionality, its elements, and how they interact with each other. The purpose to conduct top-down analysis is to cope with system complexity and dig down for details when required, which is often supported by expensive and time consuming downstream communication and coordination with other internal or external developers of the organization. Therefore, our approach to construct execution views aims to support top-down analysis providing means to collect and analyze system-specific high-level runtime
information first, and then zoom in on details when needed easing downstream communication and coordination.

Figure 1 gives an overview of our approach, its inputs, workflow, and output. The input is composed of a set of pre-defined execution viewpoints, an execution metamodel, a description of the construction requirements, and the source of runtime information. The set of pre-defined execution viewpoints and the execution metamodel act as blueprints that describe the runtime perspective on a system of a specific development organization. Understanding this perspective up-front is important to identify the requirements to construct an execution view for a software-intensive system developed by the specific organization. The construction requirements should include at least two items. First, a set of execution scenarios that were identified as representative or key scenarios for the problem that triggered the construction activity. Second, a list of stakeholders that includes the practitioners and other personnel of the development organization who need to be involved in the construction as sources of support, information, validation, and ultimately to use the constructed view. The source of runtime information is composed by the system log file and runtime measurements, which are collected from running the set of key execution scenarios, identified in the construction requirements, with the assistance of some of the identified stakeholders, e.g., system operators or practitioners familiar with the system.

The workflow of the approach is comprised of four activities. The first three are tool-supported activities that implement a dynamic analysis technique. This technique enables the extraction of high-level runtime information from the collected runtime data and the construction of the model(s) of a required execution view. The fourth activity is to present a constructed execution model to the stakeholders and thus capture their feedback and need for details. To obtain the final output of the approach (i.e., an actual system execution view), several iterations of the application of the dynamic analysis are necessary when a view is constructed for very first time. The purpose of the iterations is to collect the feedback of practitioners (e.g., software architects and designers) and to fine-tune the constructed model until it aligns with the construction requirements and therefore is accepted as part of an actual system execution view.
The next sections present the approach in depth. Section 4 presents the details of the input in terms of the viewpoints and the metamodel (the construction requirements are project-specific and will be discussed in section 6.1). Section 5 explores the source of runtime information and the dynamic analysis technique. Section 6 illustrates the whole approach through a case study.

4. VIEWPOINTS AND METAMODEL OF THE RUNTIME OF A SOFTWARE-INTENSIVE SYSTEM

Often in large development organizations, the concerns of the practitioners and the information to address those concerns are not explicitly described or documented. Thus, we have observed that practitioners are not particularly familiar with abstractions that allow them to efficiently describe, analyze, and understand the runtime of a software-intensive system. By contrast, they are more familiar with source code artifacts and related abstractions. For instance, Figure 2 shows a basic top-down description of a key execution scenario of an MRI system in the field. This description shows a high-level execution profile that describes the steps of the execution scenario and the required software components that implement each step. The usual approach to extend this description would be to map software components to source code artifacts such as code modules, classes, or methods. However, this can be overwhelming for systems with large and heterogeneous implementations (as we described in Section 2.1): one can easily miss relevant elements and relationships that play an important role in the runtime of a software-intensive system. In the rest of this section, we summarize a set of viewpoints and a metamodel that capture and document guidelines to describe the runtime of a large and software-intensive system.

![Figure 2. A description of the MRI system execution](image)

4.1. Execution viewpoints

A viewpoint consists of the conventions for the construction, interpretation, and use of a given system architectural view, which describes information that addresses the concern of a particular system stakeholder [1]. Some of the most popular viewpoints are the logical, process, development, and physical ones [2], and similarly the module, component-connector, and allocation viewpoint [18]. Through the development and application of our research we learned that indeed viewpoints are key elements for the construction of views [9, 10]. Therefore, as part of our research we defined a set of execution viewpoints that help us to construct execution views for the large software-intensive system of our industrial partner [4].

The identification of this set of viewpoints is based on our observations and a set of dedicated interviews with key practitioners. Table 1 summarizes the information that describes the identified viewpoints. This includes the stakeholders, a set of development activities, the involved
concerns, and the type of models that build views that conform to these viewpoints. The stakeholders are the typical kind of practitioners that are concerned about the runtime of the system. The concerns are represented as questions that practitioners have about the actual execution of the system and need to be answered for the proper planning and execution of the identified kind of development activities. The type of models classify the models that describe information to address the identified concerns. Further technical details of this set of viewpoints can be found in [4, 28], which served as input and reference for the definition of the new version of the ISO/IEC CD1 42010 standard.

The role of this set of viewpoints within our approach is to identify the requirements for the construction of a given view. The organization- and system-specific concerns and guidelines that these viewpoints describe represent an important input for our approach. These are especially useful to focus and ease the interaction with practitioners prior to and during the construction of execution views (e.g., execution profile, resource usage, and concurrency). Without execution viewpoints, the early validation of our approach required intensive interaction with experts (e.g., managers, architects, and designers) of our industrial partner. This was necessary to define the scope of the reconstruction activity (scenarios), the experts to be involved during the reconstruction (stakeholders), and ultimately the expected sort of information to be contained in the constructed execution views. Although this interaction was a repetitive and often time-consuming process, we found that this type of interactions was useful to observe, identify, and document the concerns of our stakeholders (practitioners of our industrial partner) about the actual execution of their system and the sort of information that can address them.

### Table 1. Viewpoints for the construction of execution views

| Stakeholders: | Software architects, designers, developers, testers, and system platform supporters. |
| Development activities: | System understanding, analysis of alternative designs and implementations, introduction of new hardware resources, testing, conformance of design and implementation, corrective maintenance, and tuning of nonfunctional properties. |
| Identified Concerns | Models |
| Execution profile viewpoint | What are the major components that realize a given system function? | Functional mapping |
| What are the high-level dependencies of major components? | Workflow overview |
| What are the major tasks that build the execution workflow of key execution scenarios? | Matrix model |
| What is the development team that develops or maintains a given system’s function? | Sequence diagrams |
| Execution concurrency viewpoint | Which runtime elements execute concurrently? | Workflow concurrency |
| How does the runtime concurrency match the designed concurrency? | Process and thread structure |
| What are the aspects that constrain, coordinate, and control the runtime concurrency of the system? | Control and data flow |
| What are the bottlenecks and delays of the system and their root cause? | |
| What are the opportunities to improve the concurrency of the system? | |
| Resource usage viewpoint | How to assure adequate resource usage and justify the development effort needed to accommodate hardware resources changes? | Task resource usage |
| What are the metrics, rules, protocols, and budgets that rule the use of resources at runtime? | Component resource usage |
| How software components and their respective processes consume processor time or memory within key execution scenarios? | Thread resource usage |
| Does the realization of the system implementation have an efficient resource usage? | |
4.2. Execution metamodel

The metamodel illustrated by B in Figure 1 organizes a set of concepts and relationships between them that play a role in the runtime of a software-intensive system. This set of concepts and relationships are instantiated in the type of models that build views related to the execution viewpoints described in Section 4.1 and Table 1. The organization is a hierarchical representation that links architectural concepts (e.g., execution scenario, task, software component, and relationships between them) to actual runtime platform elements (e.g., processes, threads, and their activity) and other important elements or resources (e.g., data, code, and hardware) that belong to a software-intensive system at runtime.

It is important to notice that the elements and relationships in the metamodel are not an exhaustive description of all possible high-level concepts, elements of the runtime platform, and runtime resources. Instead, we consider it as a subset that corresponds to the characteristics of the software-intensive system of our industrial partner and similar systems, which we have identified during the development and validation of our approach. For instance, looking at the introduction of our approach [8], one can notice that the metamodel did not consider the concept of processing node and interactions between software components, which are the result of further development reported in [4, 7]. In the rest of this section, we describe the current key elements in the metamodel and their relationships. In Section 6.3, we will describe how the elements in the metamodel are extended or specialized to construct specific execution views.

4.2.1. Execution scenarios, tasks, and users

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 [29]. 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. In our approach, we assume that a set of execution scenarios can represent a benchmark of the actual system execution. For this purpose, it is necessary that the development organization, based on domain knowledge, point out and agree on the key execution scenarios that compose the benchmark.

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 function. 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 and System Startup may not be too relevant. However, in a large software-intensive system, the system startup involves a wide range of interactions 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 because some tasks can be so complex that it is necessary to divide them into smaller tasks.
4.2.2. Processing nodes, software components, and processes

Large software-intensive systems often include more than one computer or specialized hardware devices. A **processing node** represents a computer or hardware device where part of the software components of a software-intensive system are deployed and executed. 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 runtime platform hosting the software of the software-intensive system. It represents a running application, including its allocated resources: a collection of virtual memory space, code, data, and platform resources. Often at runtime, a large software system is composed of many running applications that interact with each other in order to realize the system functionality.

The metamodel describes that one or more running processes make up a software component, because we take into account two types of relationships between processes. The first type comprises actual parent-child relationships between processes. Such a relationship usually happens when a main process creates another process to delegate a temporary or specific function. The second type comprises design relationships established by the development organization, to group processes and thus reduce complexity and facilitate the analysis of the system. For instance, when the number of processes is large, some relationships are established to distinguish a subset of processes from the rest, because the processes in the subset share particular functional or non-functional characteristics. The latter supports our view of software component as a set of processes because experts can then identify a set of processes as important, reusable, non-context-specific, distributable, and often independently deployable units.

4.2.3. Threads and their execution activities on resources

A process starts running with a primary **thread**, but it can create additional threads. A thread represents code to be executed serially within its process, which at the same time is the realization of **execution activities** for the utilization of various resources. These activities can be distinguished into three groups according to the type of the involved resource:

- **Data Access**: This type of activity represents the usage of different sorts of data structures. 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.

- **Code utilization**: This type of activity represents the execution/loading of code. Code includes executable code from the process executable files and from statically or dynamically loaded libraries. Executables and libraries can be distinguished either as system-specific or as provided by the runtime platform (platform API). System-specific code includes implementation elements such as libraries and code modules of the software system.

- **Platform Utilization**: This type of activity represents the utilization of platform resources. The processing nodes and the runtime platform of a software-intensive system provide hardware and software resources that the software of a software-intensive system use at runtime. Hardware resources include processors, memory (virtual or physical memory) units, and other sorts of hardware devices that the processes of the software system access using software resources like APIs and communication services. Executables and libraries provided by the runtime platform also qualify as platform resources.
4.2.4. Interactions between software components

The grouping of processes into software components and the analysis or interpretation of their execution activities enables the identification of interactions between software components. Interactions between individual processes can be identified and therefore between their corresponding software components. Based on our observations and the literature related to software architecture [2, 30], we can identify three types of interactions: (1) Data-sharing interactions that allow a number of processes to share and access one or more data repositories. (2) Procedure call interactions that are some sort of inter-process function calls or message-passing operations. (3) Execution coordination interactions allow two or more processes (or threads) to signal to each other when certain events occur, e.g., semaphores and mutexes.

The identification of instances of these type of interactions between software components, as well as instances of the other elements in the metamodel, is possible analyzing the runtime activity of the system elements. For instance, analyzing read and write data operations, performed by two different processes over a common data file can help to infer data-sharing interactions between the two processes. Then, if each process belongs to separate components, the identified interaction will represent a data-sharing interaction between the respective components.

5. DYNAMIC ANALYSIS TECHNIQUE

The approach uses a dynamic analysis technique and a source of runtime information from which it extracts, abstracts, and presents runtime information in terms of the elements described in the execution metamodel. The technique consists of three tool-supported activities (see A in Figure 1): task identification, interpretation of runtime information, and construction of execution model. These activities and the way we implemented them aim at facilitating the top-down extraction and presentation of runtime information. In this section, we describe the sources of runtime information and the activities of the technique.

5.1. Source of runtime information

As illustrated by C in Figure 1, the source of runtime information used by our approach is a combination of logging and runtime measurements. Logging is a feature often implemented as part of large software systems to record and store information of their specific activities into dedicated log files. Log files are often specifically created and managed by the development organization to support developers, testers, and other specialized users with development activities such as debugging, testing, and corrective maintenance [31, 32]. Log files can also contain messages about the workflow of the system functionality as described in Figure 3. Runtime measurements include data about the runtime activities of processes and resources such as processors and memory of the system’s processing nodes. Process activity and resource activity belong to monitored information (see Section 2.3). Most runtime platforms offer tools and facilities to collect runtime measurements. For example, process activity and resource activity can be monitored with tools like Process Monitor in the Microsoft Windows platform [33]. Other tools are also available to monitor process activity in the Linux and Unix platforms [34]. Runtime measurements are available in semi-standardized formats independently of implementation technologies 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).

The main advantage of runtime measurement and logging is that they are less intrusive than techniques used to collect execution traces (often used to construct module views, see Section 2.1). We attribute this to two factors: First, most runtime platforms provide monitoring facilities that aim to produce negligible overhead. Second, the overhead produced by logging is often part of the normal system behavior or within the expectations of the development organization. In addition, having logging and runtime measurement tools running at the same time eases the collection and synchronization of runtime data. For instance, Figure 4 shows that when a log message is written in a log file, the tool that monitors process activity, besides other events, records the write event that happens on the log file, including information such as the writer process, the size of the message, and the timestamp of the write event. We exploit this fact to synchronize the logging and other runtime measurements used in our approach.

The mapping between B and C in Figure 1 illustrates how logging and runtime measurements can provide information to describe the runtime of a system. In particular, the technique uses workflow messages, like those illustrated in Figure 3, to identify high-level abstractions such as execution scenarios, tasks, and software components (see Section 4.2.1). From runtime measurements, the technique uses process activity to identify elements such as processes, threads, and their activities on resources like data, code, and hardware devices (see Sections 4.2.2 and 4.2.3).

<table>
<thead>
<tr>
<th>Time</th>
<th>Subsystem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scanner</td>
<td>... Scan Starts</td>
</tr>
<tr>
<td></td>
<td>Scanner</td>
<td>... Executing Acquisition</td>
</tr>
<tr>
<td></td>
<td>Scanner</td>
<td>... End Acquisition</td>
</tr>
<tr>
<td></td>
<td>Scanner</td>
<td>... Reconstruction Ends</td>
</tr>
<tr>
<td></td>
<td>Scanner</td>
<td>... Scan Ends</td>
</tr>
</tbody>
</table>

Figure 3. Example of workflow messages extracted from a log file

Figure 4. Simultaneous gathering of runtime data from logging and process activity

5.2. Task identification

The task identification activity of the technique is to accomplish: 1) the identification of the tasks that build the workflow of an execution scenario, and 2) the synchronization of logging and runtime measurements, i.e., process activity information, for each identified task. Figure 5 shows the input and output of this activity. The input consists of the logging (e.g., a log file) and the collected runtime measurements (e.g., monitored process activity) of the executed scenario. 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 process that logs
the workflow messages. The task identification provides two important outputs. The first consists of the workflow messages that represent the boundaries of an identified task. Second is the candidate process to represent a software component, under the assumption that only important or main processes write log messages. The task identification activity is implemented as follows:
- Monitored process activity is parsed to identify write events in the log file.
- The parameters in an identified write event (offset and length) are used to extract the corresponding logging messages from the log file.
- Begin-end text pattern matching indicates if the logging message is a workflow message that starts or ends a task (in Section 5.3, we describe more about the definition and application of text patterns within our approach). This leaves two options:
  - If it is, the task 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.
  - Otherwise, other monitored process activity events and logged messages are added sequentially as execution information of the current identified task.

The implementation of the identification of the boundaries of tasks assumes that the log file contains ‘begin’ and ‘end’ messages, which are logged prior to and right after a task is executed. For situations where this 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 to split execution scenarios into tasks and synchronize logging and runtime measurements.

5.3. Interpretation of runtime information

The output from the previous activity provides partial runtime information (task and some software components) that can describe an execution scenario, but it is still missing information that describes all other possible software components and their interactions within the scenario. To cope with this, the technique implements and applies the concept of mapping rules described in [9]. The technique uses two sets of mapping rules: One set to abstract logging messages (LMR) and another set to abstract runtime measurements, i.e., process activity events (PAMR). Figure 6 illustrates the concept of mapping rules and how both LMR and PAMR as partial functions because they do not necessarily abstract every logging message (L_i) or every process activity event (A_j) into an execution interaction (E_k).

An execution interaction E is a tuple with instances of the elements in the execution meta-model: A subject represents a software component or a process entity; a verb represents an execution activity (e.g. read, write, load, and execute); an object represents data, code, platform resources, software component, and process; info may contain additional information such as the thread ID. In general, the implementation of a mapping rule looks for text patterns in a logging message (L_i) or a process activity event (A_j) respectively and generates one or more execution interaction tuples. For example, the application of a mapping rule to a process activity event that describes a running Process B creating another process B1 within its thread T, generates the tu-
ple: \textit{(Process B, Create, Process B1, Thread T)}. The task identification activity, described in Section 5.2, also uses part of the group of mapping rules to identify workflow messages.

![Figure 6. Concepts for interpretation of execution information](image)

Figure 6 illustrates in overall the interpretation of runtime information where the sets of mapping rules are applied to the logging messages and process activity events bundled in a task (source view) to abstract them into execution interactions in terms of the high level abstractions of the metamodel (target view). The output of this activity is a set of interaction tuples, which we store as a graph structure (interaction graph) to facilitate further analysis and manipulation of the runtime information. An additional output is a list of the identified software components and other system resources within the task.

![Figure 7. Interpretation of execution information of a task](image)

5.4. Construction of execution model

This activity of the technique focuses on the construction of an execution model querying the runtime information stored in the interaction graphs of an execution scenario (see Section 5.3). The querying focuses on finding information that addresses the concerns that triggered the construction of the view. The concerns and the sort of runtime information that may address them are described as part of the execution viewpoint (Section 6.3) driving the construction. This viewpoint also provides guidelines about the notations and the diagrammatic or graphical representations that can be used to construct execution models. The output of this activity is the constructed execution model, which is presented to the respective stakeholders in the model presentation activity of the approach. In Section 6.3, as part of the validation of the approach, we illustrate in more detail the execution model activity with some examples of actual execution models.
6. VALIDATION OF THE APPROACH WITH THE MRI SYSTEM

The application of our approach has been focused in the construction of various execution views for the Philips MRI system, e.g., reported in [7, 8]. This system is a large software-intensive system in the healthcare domain. Figure 8 illustrates and summarizes the size and implementation technology of this system, which demonstrates its complexity and importance as a case study. In this article, we illustrate the application of our approach (Section 3) with the construction of an execution profile view for the key execution scenarios of this system. The rest of this section describes the details of: the aspects of the construction input, the execution of the dynamic analysis technique, and the technical findings of constructing and using the constructed execution profile view in practice.

![Figure 8. Philips MRI system size and implementation technology](image)

6.1. Construction input

The construction input is composed of the construction requirements and the runtime data. The construction requirements, as we introduced in Section 3, include at least the key execution scenarios and the stakeholders related to the problem at hand. To construct the execution profile view, the key execution scenarios are those that are often found within the planning and execution of usual development activities (listed in Table 1). The stakeholders include software architects and designers related to each execution scenario. For each scenario, at least one of the stakeholders was identified as the problem or domain expert. The other involved stakeholders were mainly identified as practitioners interested in the design, implementation, and documentation of a solution of the problem at hand, who need to be contacted for additional information, help with the execution of the involved scenarios, or validation of the constructed views; and hypothetical views or sketches. The sketches, often made by the problem-domain expert, were mainly initial guidelines to discuss and identify which concerns (see Table 1) need to be addressed, and therefore define the course of the application of the approach.

The runtime data was collected executing the scenarios identified in the construction requirements. To collect the data for the construction of the execution profile view, we often needed the cooperation of an additional stakeholder identified as an end-user or system expert that can operate the MRI system running the chosen execution scenario. Then, we collected the logging file created by the MRI system; and since the runtime platform of this system is Microsoft Windows, we used monitoring tools like the Process Monitor Tool [33] to collect events...
about file system activity, process activity, and access to the Windows Registry. Table 2 summarizes the sort of data collected from the MRI log file and the runtime measurements for the construction of an execution profile view.

<table>
<thead>
<tr>
<th>Collected Data</th>
<th>Extracted Information</th>
<th>Source of Text Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow messages</td>
<td>Tasks, Software components, Processes, and Threads</td>
<td>MRI Log Documentation</td>
</tr>
<tr>
<td>Debug messages</td>
<td>Major code modules</td>
<td>MRI naming conventions</td>
</tr>
<tr>
<td>File access events</td>
<td>HW &amp; SW configuration and setting data</td>
<td>Filesystem structure and [35]</td>
</tr>
<tr>
<td></td>
<td>System database data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DLLs, assemblies, and dynamic wrappers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Script programs</td>
<td></td>
</tr>
<tr>
<td>Process activity</td>
<td>Software components, processes, and threads</td>
<td>[35, 36]</td>
</tr>
<tr>
<td>Windows Registry access</td>
<td>COM elements</td>
<td>[36, 37]</td>
</tr>
<tr>
<td></td>
<td>HW &amp; SW configuration and setting data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Communication services and platform resources</td>
<td></td>
</tr>
</tbody>
</table>

### 6.2. Task identification and interpretation of runtime information

Together the requirements analysis and the runtime data represented the input for the task identification and interpretation of runtime information activities (see A in Figure 1, and Sections 5.2 and 5.3 respectively) of the approach. These activities are often complemented by the feedback of the involved stakeholders. In our validation, a first feedback was used to validate the semi-automatic identification of the tasks that compose the workflow of the execution scenario at hand.

In our validation, the interpretation of runtime information includes the gradual implementation of specific mapping rules (see Section 5.3) for the MRI system. Table 2 describes the sort of runtime elements extracted from the MRI system logging and runtime measurements. This description also lists the consulted technical information or documentation to identify text patterns used by implemented mapping rules to construct the execution profile view for the MRI System. A set of mapping rules abstracted naming conventions and other patterns of the filesystem structure to group and classify runtime elements. This enabled the identification of data configuration files and the extraction of information that describes the runtime use of configuration data of specific hardware devices (see Section 6.3.1 for more details). Another set of mapping rules were to interpret debug messages in the MRI log file, which provide patterns to identify calls to major code modules. Some more specific mapping rules abstracted text patterns in the events accessing the Windows Registry [37] to identify COM classes and interfaces used at runtime.

Implementing the last set of mapping rules, we observed that the extracted information was implicitly presented as specializations of the elements in the metamodel. Figure 9 illustrates how persistent data is specialized into data files and then configuration and data repository elements. Code is specialized into system-specific code and then into COM elements. Finally, resource elements are specialized into platform resources as communication services to extract information that describes communication dependencies between software components and their respective processes.
6.3. Construction of models for an execution profile view

An execution profile view is a set of models that provide overview and facilitate the description of details about the execution of the functionality of a software-intensive system (represented by a set of key execution scenarios identified by the system’s development organization). The information described by the models of an execution profile view represents tangible evidence to initiate and focuses the top-down analysis and discussion of a problem at hand addressing the concerns of an execution profile viewpoint (see Table 1). Initially, as we presented in [8], our approach provided two types of execution models: scenario overview (graph-based view) and matrices, which we have refined and extended with the further development of our research as execution models of an execution profile view. Currently, an execution profile view consists of two types of models: overview models and models to dig down for details. These two types of models ease the analysis of execution information in a top-down fashion, i.e., starting with high-level description or overview to detailed descriptions.

6.3.1. Overview Models

Overview models are tangible assets that describe the actual execution of the system functionality at an overview level, especially without being overwhelmed by the size and complexity of the system at hand. Examples of this type of model are functional mapping and workflow overview models.

A functional mapping model, shown in Figure 10 and A in Figure 13, is a graph-base representation that describes relationships between high-level elements of an execution scenario. The notation of this model can be described as follows:

- The scenario is divided in a set of tasks that are linked to the software components that realize each of them. The links, e.g. color-coded edges, should identify the trajectory of a task through the rest of elements in the model.
- Each software component is respectively mapped to its set of running processes (e.g., using record structures where the fields of the record represent the respective processes).
- The links continue and describe the activities of a software component (acting as the subject) on objects; the objects represent the identified specializations of data, code, or platform resources, e.g., components may read from or write on data repositories.

The functional mapping model illustrated in Figure 10 is a description that we constructed to analyze the management of MRI coil devices (key system specific hardware devices that work as antennas to collect data from the magnetic phenomenon generated by the system). Thus, the tasks described in the model aggregate the operations that a system operator executes to add, remove, and enable the use of this device at runtime. The model describes the software components, the processes (system and platform specific) that build each component, and the data and code elements that are involved at runtime to enable the management of a coil device. The main
concerns of the stakeholders interested in this scenario included: what are the major components that realize the handling of MRI coil devices? And what are the high-level dependencies that couple those major components? Especially dependencies related to the coil configuration data which was subject of change within the given development project.

A workflow overview model, shown in Figure 11, is a representation that describes how the identified tasks of an execution scenario are actually executed (sequentially or concurrently) across the various processing nodes of the MRI system. One can consider this model as a Gantt-chart-base description with the following notation:

- The task of the scenario are represented as segments horizontally and vertically distributed.
- The horizontal organization corresponds to a time axes that represent the time when the task is executed, and its length the time duration of the respective tasks.
- The vertical organization corresponds to location axes, which represent the processing node where the task is executed.
- Color-coding can be useful to distinguish the function or nature of the involved tasks and the borders between their locations.

The workflow overview model illustrated in Figure 11 is a description that we constructed to identify and make it explicit the actual sequence of events or tasks within the start-up of the MRI system. The information in the model describes the major tasks that are executed, across the different computers that compose the MRI system (Host, Recon, and DAS) and address the concern: what are the major tasks that build the execution workflow of system startup?; and if possible identify opportunities to speed up this system functionality. The major tasks described by this model represent high-level aggregations of the various events that happen from the time a system operator press the system Power On button, to the time one can actually use the system to conduct a clinical scan. The current value of this model, as reported by our industrial partner, is that it represents a tangible benchmark to monitor changes in this key scenario through ongoing and future development project.
6.3.2. Models to dig down for detailed information

Overview models present execution information at a high-level; however, high-level descriptions often need to be justified or explained with details to address the stakeholders’ concerns following top-down analysis. We observed that some of the usual reasons to look for details are when practitioners are not so familiar with the scenario at hand, or they find that high-level descriptions drift from their mental models. Often, mental models either drift from the constructed models due to changes on the system or miss implementations of the designs. Thus, it is necessary to complement overview models with models that provide detailed description, but still without being overwhelmed by the size and complexity of the system. Examples of these models that enable this are matrix models and sequence diagrams.

A matrix model, shown in Figure 12, and B and C in Figure 13, is a description of execution information that helps in the identification of the details that may create relationships between major runtime elements and may be characterized as dependencies. The details that are analyzed to determine the relationship between runtime elements are described in the cells of the matrix models. These details are presented for analysis in a vertical and a horizontal dimension.

- In the vertical dimension, a matrix model describes the relationships between two different types of runtime elements of an execution scenario, e.g. Figure 12 shows a matrix models that describe the relationships between the tasks of an execution scenario and its software components. The selected information detail is the number of active process and threads of the software components within the given tasks.

- In the horizontal dimension, a matrix model describes the relationships between elements of the same type. For instance, the matrix model B in Figure 13 describes the relationships between the tasks of an execution scenario looking at their common used data elements.

| Table 3. Matrix models for runtime information |
|--------|--------|----------------|
| Rows   | Columns| Cells          |
| I      | Software Components | Software Components | Data, code, and platform elements |
| II     | Tasks           | Tasks          |
| III    | Software Components | Tasks | Component’ interactions |
| IV     | Software Components | Tasks | Component’ processes |
Table 3 lists the current combinations supported by the matrix models constructed with our approach. The correspondence between graph-based and matrix-based representations is a usual practice and often used to describe and analyze complex systems [38, 39].

![Figure 12. Example of matrix models for execution information analysis](image1)

![Figure 13. Example of zooming in details with matrix models](image2)

In the application of our approach, we used matrix models to complement the information described by functional mapping models. For instance, the dependency matrix models shown in Figure 12 constructed to complement the functional mapping model in Figure 10 and dig-down for details in the runtime handling of the MRI coil device. The reasons to use matrices were to describe and determine which software components were actually active within the execution scenario. To do so, we looked at the number of active process and active threads per component.
Similarly, the matrix models shown in Figure 13 complement the functional mapping model shown in the same figure. These models were constructed to describe the usual execution scenario of a clinical scan using the MRI system. The matrix model B in Figure 13 helped to identify the UI Menu Structure elements commonly accessed in the execution scenario by two of the key MRI system components (it is important to notice that these models were simplified filtering out information of some software components that were not in the scope of the case study). The matrix model C in Figure 13 helped to identify the specific Configuration Repository element commonly used or read by the SCANNER component in the execution scenario.

**Sequence diagrams** are typical representations of execution information, particularly for the reverse engineering of object-oriented systems [11]. As part of our approach, we experimented with the construction of sequence diagrams especially to describe details about the time dimension and temporal relations that may exist within an execution scenario. Figure 14 shows examples of sequence diagrams that we constructed to zoom in on details. To construct sequence diagrams, in particular for complex and large execution scenarios, implicit input from the interested stakeholders is needed. This input should be collected during the identification of the construction requirements and should contain at least the following information:

- The tasks or the relevant parts of an execution scenario that concern the stakeholders.
- The types of execution entities, if possible the specific software components, of interest to be described in the constructed sequence diagram.
- The type of relationships or interactions of interest to be described in the sequence diagram, e.g., data sharing, procedure calls, execution coordination, or other specific detailed interaction.

Capturing these requirements from the stakeholders can be often challenging, especially when practitioners are used to draw their mental models using boxes and arrows rather than the granular notations such as sequence diagrams. Nevertheless, capturing these requirements helps to understand the experts’ needs and to construct sequence diagrams to scope the analysis and dig down for details in the information described by overview models.

The sequence diagrams presented in Figure 14 were constructed to complement the description and analysis of the runtime handling of the MRI Coil device described by the overview model in Figure 10. The only initial requirement that we could capture at the beginning was the task of interest, the Add Coil task. With this input at hand, we constructed the first sequence diagram, A in Figure 14. The elements for this sequence diagram were chosen as the software components (SWC), groups of the main code artifacts (SW modules and COM components), and groups classifying the data configuration. Although this first sequence diagram was not optimal (mainly due to the amount of information and readability), the domain expert of the scenario was able to annotate this model identifying five sub-tasks: operator actions (1), COM communication (2, 4), load of available Coils’ configuration (3), and update system repository configuration (5). The further analysis on this sequence diagram and the scope of the case study triggered the construction of a second sequence diagram, B in Figure 14. This time the input for the construction included the subtask 3 as section of interest (a key aspect of the scope of the case study), the main software modules, and the coil configuration data group as the interesting runtime entities. The constructed sequence diagram helped the practitioners to zoom in the actual interactions caused by the realization of the load of available Coils’ configuration subtask, using actual runtime entities and relationships that most other techniques constructing sequence diagrams do not describe. Currently, to construct these sequence diagrams, we use scripts to query the execution informa-
tion stored in interaction graphs creating textual descriptions of sequence diagrams that are pro-
cessed by the UMLGraph tooling [40].

Figure 14. Example of zooming in details using sequence diagrams

6.4. Major technical findings constructing and using an execution profile view

Some of the main technical findings that we observed constructing and using an execution profile view are:

- We found that the combination of logging and process activity is a useful source of information to construct an execution profile view, especially for heterogeneous implementations. This also showed our industrial partner some added value for the infrastructure in place for collecting data over years of development.

- The involved practitioners appreciated the new structured process supported by the use of viewpoints and a metamodel, and a set of models to present runtime information in a top-down fashion. For instance, 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 based on the collected runtime data.

- Using the constructed view, we observed that execution models were considered as tangible evidence. On the one hand, this evidence refreshed and validated the mental pictures used by various experts. On the other hand, this evidence helped some senior practitioners to convey the understanding of the system to other practitioners, such as designers. For instance, some practitioners, who are mainly familiar with the system functionality, were able to map the system functional decompositions to actual runtime entities using overviews models.
Different aspects of the realization of the implementation were uncovered and made explicit using overview models. Some practitioners highlight that describing the mapping of processes to software components (shown in Figure 10 and Figure 13) helped them to identify the function of a software component, its implementation technology, and whether it depends on platform utilities. In Figure 10 and Figure 13, we can identify software components containing \texttt{csc.exe} processes, which represent .Net implementations. Similarly, we can identify software components containing \texttt{attrib.exe} and \texttt{unzip.exe} processes, which represent components that require platform utilities to manage access rights and data compression respectively.

It was possible to identify the use of some design and implementation policies of the development organization. Some communication paths such as the usage of Windows Registry entries are being removed from the implementation, using overview models, some experts were able to spot desired and undesired communication paths. With similar models, it was possible to recover many implicit concepts in the actual utilization (distribution and combination) of configuration data and major code artifacts within the key execution scenarios.

Our initial goal to construct matrix models was to provide detailed information, such as quantitative information about commonalities between runtime elements. In addition, some practitioners found that describing the details of relationships between software components and tasks (e.g., using the number of active processes or threads as illustrated in Figure 12) was useful to assess the overhead created by the implementation practices and the utilization of third-party components in some of the key execution scenarios of the MRI system.

7. DISCUSSION OF THE PROPOSED APPROACH

In this section, we discuss three different aspects of the proposed approach: a) suitability of our approach as a reverse architecting solution with respect to an accepted framework; b) the specific added value of the dynamic analysis technique with respect to other related techniques; c) limitations of the approach and its validity.

7.1. The approach as a reverse architecting solution

We demonstrate that our approach qualifies as a reverse architecting solution (in particular to construct execution views of large and complex software-intensive systems) by mapping it to Symphony [9, 10], a conceptual process framework for architectural reconstruction. Two main points demonstrate the correspondence of this framework and our approach. First, the Symphony framework is an amalgamation of common patterns and best practices of reverse engineering using architectural views and viewpoints. Our approach follows the same principle by aiming to construct execution views (e.g., the execution profile view in Section 6.3 and a resource usage view [7]) using pre-defined execution viewpoints (see Section 4.1 and [4]) and an execution metamodel (see Section 4.2).

Second, the Symphony framework defines architecture reconstruction as a process conducted in two explicit major phases, reconstruction design and reconstruction execution. Reconstruction design is useful beyond the scope of the construction of a particular view, because it plays a role in continuous architecture conformance checking and the construction of other views. In our approach, we adopt this aspect starting with the identification of the viewpoints and metamodel to capture the runtime perspective of a specific development organization, including their concerns.
and how to address them in general (see Section 4.1 and [4]). For the construction of a particular view, reconstruction design includes two main aspects, problem elicitation, and concept determination. Problem determination is about the discussion and analysis of the problem that trigger the reconstruction with the system stakeholders. Concept determination consists in the identification of the relevant architectural concepts to address the problem at hand. In our approach, we conduct problem elicitation and concept determination interacting with the practitioners for the identification of the construction requirements (described in Section 6.1).

Reconstruction execution yields the architectural view needed to solve or address the problem that triggered the construction of the view through an extract-abstract-present process. In our approach this matches the activities of the dynamic analysis technique. The extract part is about collecting the data to construct the view, which in our approach is performed collecting the runtime data (see Section 5.1) for the selected key execution scenarios (see Section 6.1). The abstract part consists in extracting or inferring information from the collected data, which in our approach match to the task identification (see Section 5.2) and interpretation of runtime information (see Section 5.3 and Section 6.2) activities. Finally, the presentation part consists in making the extracted information accessible to the stakeholders in the form of a model of a constructed view as in the model construction activity of our approach (described in Section 6.3).

7.2. The value of the dynamic analysis technique

We consider that the dynamic analysis technique used by our approach has several characteristics that help to overcome practical issues in constructing views for large and complex software-intensive systems like the MRI system. Table 4 lists these issues and how our technique and the related techniques presented in Section 2 address them.

Overhead: An important requirement of development organizations of large software-intensive systems is that the application of reverse engineering techniques should generate the least possible overhead. On the one hand, it is appreciated if development activities do not require extra effort from practitioners for the application of a given technique. On the other hand, it is especially appreciated that the execution of the system does not change its actual temporal relationships and performance due to the application of a given technique. For instance, we consider that code-based techniques (Section 2.1) generate severe overhead, due to the effects and required effort to instrument the source code to collect data that can capture the relations between relevant runtime elements (Section 4.2). By contrast, techniques to construct management views (Section 2.3) are less intrusive because most of them analyze runtime data generated by the system itself or its runtime platform without requiring dedicated instrumentation. In the case of the technique of our approach, it incurs minimal overhead using already available data from logging and runtime measurements. As we described in Section 5.1, logging is a source of runtime information created and maintained by development organization. Although it is based on a sort of source code instrumentation, our technique has not required extra instrumentation for logging so far. Instead, it complements the existing logging, with runtime measurements collected with monitoring tools provided by the system runtime platform.

Heterogeneity: Our focus is on large and heterogeneous software-intensive systems, thus it is desired that the used technique can abstract away or cross the borders between different programming languages and paradigms often used in the implementation technology of this type of systems. On the one hand, as we described in Section 2.1, current code-based techniques do not
provide guidance on how to integrate and analyze execution traces of heterogeneous systems. Thus, their support for heterogeneity is relatively low. On the other hand, the heterogeneity of techniques to construct management views (Section 2.3) is high because they analyze major system elements as black boxes, which abstracts away the system implementation technology. The case is similar for the technique of our approach, which deals with heterogeneity constructing top-down descriptions. At the top, it describes software components as a set of processes abstracting away their implementation language and paradigm. However, it also provides means to describe and distinguish aspect of the implementation and realization, as we described in Section 6.4 and illustrated in Figure 10 and Figure 13.

Architectural abstraction: Development organizations of large software-intensive system often require abstraction and system decomposition to follow top-down analysis along development activities. On the one hand, code-based techniques only extract and present high-level abstractions in terms of code artifacts. On the other hand, techniques to construct management views (Section 2.3) directly extract and present major system components. We consider that the technique of our approach is at the architecture level of abstraction, since it describes the execution of the system in terms of architectural concepts i.e. scenarios, tasks, software components, and processes (see Section 6.3).

Traceability: Although system high-level decompositions and abstractions are useful to analyze and understand large and complex systems, within a system development cycle it is always necessary to link high-level abstractions with low-level implementation artifacts (e.g. functions, methods, and classes), and vice versa. In this aspect, code-based techniques provide high traceability because it is possible to map the extracted high-level abstractions to the source code entities (e.g. function or method) tracked by an execution trace. By contrast, techniques to construct management views (Section 2.3) have low traceability because their analysis of major system elements as black boxes limits the mapping of major system elements to low-level implementation artifacts. We consider that the traceability of our approach is high for two reasons: first, we can map our abstractions to important implementation or code artifacts (e.g. main code modules, shared libraries, and COM components); second, the traceability of our approach also can span low-level data and hardware elements as it is illustrated in Section 6.3.1 and Section 6.3.2 respectively.

<table>
<thead>
<tr>
<th>Techniques For</th>
<th>System Overhead</th>
<th>Heterogeneity</th>
<th>Runtime Abstractions</th>
<th>Traceability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic module views</td>
<td>Severe</td>
<td>Low</td>
<td>Code modules</td>
<td>High</td>
</tr>
<tr>
<td>Component-connector views</td>
<td>None/Minimal</td>
<td>High</td>
<td>Components, connectors, interfaces, and ports</td>
<td></td>
</tr>
<tr>
<td>Management views</td>
<td>Minimal</td>
<td>High</td>
<td>Subsystems, applications, and services</td>
<td>Low</td>
</tr>
<tr>
<td>Our technique</td>
<td>Minimal</td>
<td>High</td>
<td>Scenarios, components, processes, code modules, and hardware and data resources</td>
<td>High</td>
</tr>
</tbody>
</table>

7.3. Potential limitations

The usage of viewpoints and metamodel, specific to our industrial partner, can be seen as a limitation for the generalization and reuse of our approach in other systems and projects. How-
ever, we do not consider that our viewpoints can be used off-the-shelf but should be customized
to the organization and project at hand to construct useful execution views in practice. The presentation of our approach in this article and our related research [4, 7, 8] illustrate how other practitioners and researchers can perform such customization for their particular settings. It may include the extension or specialization of the described execution models and execution viewpoints, adding their particular concerns and favorite ways to address them as we described in [4]. Then the execution metamodel may include similar high-level but different elements, that describe the runtime platform and the resources of their particular system. This may include changing the mapping of software components to the representative runtime element of their system runtime platform. For a single process, but still multithreaded software application, software components may be mapped to persistent threads or major objects for the case of an object-oriented implementation. For even larger and distributed systems using service oriented architectures, the service element may be included as a high-level element above or instead of software component. Our predefined set of viewpoint is referred as representative examples in the new definition of the ISO/IEC CD1 42010 standard and we welcome proposals from the community on how to customize them for other purposes.

In Section 6.2, as part of the validation of our approach, we described how the construction requirements, the system domain, and the system runtime platform are key factors that drive the implementation of the set of mapping rules as partial functions. This means that we selectively implement and apply mapping rules to a given subset of logging messages and process activity events rather than to all the collected data. Although this usage of mapping rules may look as a limitation for the completeness of our approach, we consider this as the key factor that enables the top-down approach where we aim to provide high-level information at first, and dig down for details when it is needed or triggered by the construction requirements. Another possible limitation is the fact that in the validation, the technique of our approach uses mapping rules that were implemented for the specific format or patterns of the MRI system logging and the collected runtime measurements. However, we consider this more as an illustration on how to implement and use the concept of mapping rules. Thus, other practitioners or researchers can consider those to deploy our approach in different settings.

Finally, due to the settings of our research, the perception of the value and limitations of our approach are based on our observations and the feedback of the practitioner of our industrial partner. This can represent a major limitation that we still need to evaluate with the collaboration of external researchers and practitioners willing to deploy our approach in different settings involving large and complex software-intensive systems.

8. Conclusions and Future Work

The contribution of our approach is centered on two points. First, it is a structured and problem-driven reverse architecture solution. The involved stakeholders, software architects and designers, considered important the use of viewpoints and a metamodel as guideline for the preparation of the construction requirements and the presentation of information in a top-down fashion. This allows us to conduct the construction involving the practitioners, deal with complexity, and construct useful views to address a concrete problem. Second, the stakeholders involved in the various applications of our approach got actual information about the runtime behavior and structure at an architectural level without being overwhelmed by the complexity of the software
system. These aspects make our approach an extensible, scalable, and transparent reverse architecting solution to construct execution views. In our future work, we aim to report further validation of our approach in the large and complex software-intensive system of our industrial partner and similar systems. Finally, we consider that our approach is complementary to the existing techniques, therefore in future work we aim to provide the means to link our approach with the existing techniques. This will allow the development organization to have complete and actual information about its software-intensive system.

Acknowledgment We would like to thank the Software Architecture Team and the software designers of the MRI system in Philips Healthcare, in particular Krelis Blom, Danny Havenith, and Sergiy Bachynskyy. We also extend our gratitude to Pierre van de Laar and our other Darwin colleagues for their feedback and joint work. We also thank Rob van Ommering, Frank Wartena, and the anonymous reviewers of an early version of this article.

References

ANNEX A. OBSERVATIONS FROM PREVIOUS REVIEW

This annex presents a summary of the major observations from a previous review of this paper.

Reviewing: 1

I was not convinced by the 'evaluation' section (section 8). This was largely reiterating the initial justification presented earlier. An evaluation should not try to rhetorically make points, but present evidence, such as from data gathered about the superiority of the approach. However, for a practical paper like this I think the results speak for themselves. I would therefore simply call the evaluation 'Discussion'.

Reviewing: 2

Viewing the paper stand-alone, it represents an interesting contribution for potential publication at JSME. However, having read the published paper at WCRE 2008, I feel that the delta between these two papers is rather marginal. There explanation how the submitted paper extends the previously published paper should be made prominent in Introduction or Related Work section. Further, since many parts have been just copied (more or less one to one) from the original paper, these extensions have to be made explicit. The clear deltas to the original paper are mainly Section 6 and 8. While Section 6 explains the notation, only added value is given in Section 8, the evaluation of the approach.

While the idea of an evaluation of the approach is nice, there are some questions arising:
Which code-based and application management did you take into account for table 5? All mentioned in Section 2? Is it valid to compare a group of different approaches to your approach? And do really all approach in one group score equally? How did you determine high and low (and what other scorings are possible)?

In summary, I recommend to reject the paper for the special issues because I do not see enough new material that justifies a journal publication.

Reviewing: 3

- I disagree with this statement "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. " Software engineers will need lower level information to accomplish the maintenance tasks at hand. Think of somebody who wants to detect a defect in a system. A mapping to source code is not sufficient. This person needs dynamic information to understand the cause of the defect. The problem is that as software engineers dig down run-time information, they will need to deal with the size explosion problem. Ref [10] and other papers written by the same author lead the work towards trace summarization and abstraction techniques, which in my view are crucial in the context of dynamic analysis.
- Many instrumentation techniques are low overhead and not intrusive (see Lttng). I don't think that this aspect of the work is really a contribution.
- I am not quite convinced that heterogeneous environments require a whole new approach for applying dynamic analysis. They require new views but other views, system-specific, are also needed.
Weaker points
- structure of the paper can be improved
- very weak validation
- 30% delta questionable

Detailed remarks

My first major remark is that you should do a better job at explaining:
- what your goal is
- what your scientific contributions are If I should say something about your goal now, I would pick the following sentence from the intro: "In this paper, we present a dynamic analysis based approach that recovers and analyzes information from the actual execution of a large software-intensive system". Then you go into technical details and finally you say something about "the identification and description of dependencies between high-level execution entities such as scenarios and software components. Try to make a short, strong statement about what your technique is trying to accomplish. This is currently missing as the info is spread out all over the place.
- Continuing... what are your scientific contributions? Make this very clear, because this again disappears into your text.
- Another major problem that I have... I am not sure as to how generalizable your technique is. You heavily rely on logging messages and process activity events. While I agree that the latter are constant for the OS, the former are not and are very much dependable on the organization and/or development team. Do they use logging events, is there a fixed format for this, does everybody stick to the format, etc. On another note, in some of my previous research I encountered a company that was using logging, but due to the overhead that it generated, the system just couldn't handle it... so they switched it off... Also think of channels, because this logging is quite often done at multiple levels (think of log4j).
- So, my concern is actually, did you guys think of these generalizability problems... because I would like to have seen a 'threats to validity section', detailing these potential problems. This is something that I find crucial in good journal paper...
- Your statement: "On the other hand, it is hard and expensive to obtain and integrate code-based information (execution traces) from heterogeneous systems". I am not completely convinced of this. Take a C++ and Java system. You could easily write an AspectJ and AspectC++ aspect that traces information for you and writes everything down in a trace file that has a very similar -- or identical -- structure. If you still believe this is a major issue, please explain in more detail or provide a reference.
- Section 2.1: Code-based methods... In the 2nd paragraph you state: "Then there are methods that analyze execution traces". Aren't execution traces concerned with dynamic information? Yes, they relate to code artifacts, but they are not code-based methods! So either change the title or subdivide section 2.1.
- Section 2.1: "In the literature, implementations of code-based methods are presented for specific programming languages." While this might be true for a number of techniques, this is for example not true for the work of Greevy, e.g., CSMR'05 or Zaidman (CSMR'05 & CSMR'06).
- You might want to refer to a recent survey on dynamic analysis in Section 2.1 by Cornelissen, van Deursen, et al.
- Section 2.1 seems very much technique-oriented, while section 2.2 is goal-oriented. Maybe you should try to level things?
- Your remarks that you observed that software architects work in a top-down way... have a look at the work of Von Mayrhauer or the work of Vans. A reference to that work would be very good here.
- Why can communication between SW components not be expressed in terms of methods? High-level sequence diagrams work like this, no?
- Concerning the meta-model. Did you look whether such meta-models already exist?
- What I don't understand in your figures 5 and 6: In Figure 5 you suggest that logging messages and process activity are written away in the same log file, while in figure 6 it becomes clear that those are 2 separate log files...
- Are log files kept per thread of per process? This might make a difference for concurrency (--> Heisenbugs). You fail to explain this, yet, this is very interesting!
- Section 4: "... not particularly familiar with abstractions that allow them to efficiently analyze and understand the execution of a software system"... while you are right, a recent experiment by Cornelissen presented at ICPC'09 suggest that this shouldn't be a hindrance, on the contrary even...
- What I particularly disliked in Section 6.3 is that you introduce the technique, i.e., sequence diagrams, and that you immediately also perform the validation of it, in the same section. It just looks like you quickly wanted to put this section in, without too much hassle. In doing so, I also didn't quite get what 'task 3' is.
- Section 7.3. While in previous sections you have detailed what your technique is and how it works, now you are saying "... enabled the quick identification of characteristics such as the implementation technology and the usage of third party or platform utilities..." How can I bridge this? In other words, how does your tool enable this? I haven't seen any (anecdotal) evidence on this... which you should deliver in my opinion. How do your views help you in finding this info? Point to figures, explain, please!
- I am now only giving the example for this particular item, but there are more instances of this 'bug'...
- Section 8: you might want to relate the temporal relationships to Heisenbugs (also known as the observer effect or the probe effect)
- Section 8: "by contrast, methods for application management are less intrusive because most of them analyze execution information generated by the system itself or its runtime platform". Provide a reference, findings or something else, because this is not necessarily true in my opinion (also look back at my earlier remark and think of the amount of info that is collected...).
- Your references miss information, mostly missing page numbers and publisher information.
- Now, considering the 30% delta of your paper... I agree that section 8 is new and could actually improve the paper (if executed well, see above). On the other hand, I don't agree that Section 6 is completely new, because it contains parts of the previous section 6.3. Furthermore, the section on sequence diagrams is, while new, very weak. It seems you put that section in, but you don't really come back to the usefulness of sequence diagrams later on. Furthermore, you immediately discuss sequence diagrams in the same section instead of deferring this discussion to an experimental section, which is also subject to improvement if you ask me.
- Perhaps even more important is that I think the validation is weak. It seems you are telling us some of the things that the experts liked, but you are not telling your audience how exactly your technique enables this (also see one of
my previous points). So, in my opinion, you should strengthen your validation section, by, e.g., providing anecdotal evidence on how you and/or the experts got to some specific kind of information. I do understand that you might not be at liberty to divulge MRI-specific data, but I think that you can abstract away from this, and still convey a lot more information to the reader. Also concerning the validation, please include a threats to validity section!

For me, this paper is either a very, very major revision or a reject... I leave that up to the editors.