Chapter 3

A Top-Down Approach to Construct Execution Views

Published as: Trosky B. Callo Arias, Paris Avgeriou, Pierre America – *"Analyzing the actual execution of a large software-intensive system for determining dependencies,"* In: Proceedings of the 15th Working Conference on Reverse Engineering, pp. 4958. IEEE Computer Society, 2008.

In revision as: Trosky B. Callo Arias, Pierre America, Paris Avgeriou – *"A Top-Down Approach to Construct Execution Views of a Large Software-Intensive System,"* Journal of Software Maintenance and Evolution: Research and Practice, December 2010.

Abstract

This chapter 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 approach represents an architecture reconstruction solution based on a metamodel, a set of viewpoints, and a dynamic analysis technique. The metamodel and viewpoints capture the conventions to describe and analyze the runtime of a software system developed by a particular organization. The dynamic analysis technique is 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 scanner. Therefore, the approach represents a solution that can be applied with similar large and complex software-intensive systems.

3.1 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 typically 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 led to a demand for architectural descriptions that can help software architects and designers to get a global understanding of a software-intensive system.

The ISO/IEC 42010 Standard (ISO/IEC 2007) mandates that the architectural description of a system is organized into multiple views. An architectural view (or simply, view) consists of one or more architectural models, which in turn represent a set of system elements and relations associated with them. Each such architectural model conforms to a model kind, which is part of the conventions and methods established by an associated architectural viewpoint. An architectural viewpoint frames particular concerns of the system's stakeholders and consists of the conventions for the construction, interpretation, and use of an architectural view. One of the most popular examples of viewpoints are the logical, process, development, and physical viewpoints (also known as the 4+1) proposed by Kruchten (Kruchten 1995).

As part of our research project (van de Laar et al. 2007), we investigate how to improve the evolvability (i.e., the ability to respond effectively to change) of software-intensive systems studying a Magnetic Resonance Imaging (MRI) scanner developed by Philips Healthcare MRI (Philips Healthcare 2010). In this context, we observed that up-to-date architectural views are important assets to support the incremental development or evolution of software-intensive systems. However, the architectural views of this type of systems are not always up-to-date, available, or accessible. This is especially the case when a system has a long history of being exposed to changes and contains legacy components, which are associated with multidisciplinary knowledge spread across the practitioners of the development organization. Therefore, our goal is to find methods and techniques to construct up-to-date views of large and complex software-intensive systems.

In the literature, most techniques aimed at constructing up-to-date views, focus on development or module views, typically gathering information from the system's source code (Koschke 2009). However, up-to-date module views are not enough to get a global understanding of a large software-intensive system and support its evolution. A system of this type has a large number of stakeholders, and each of them has concerns about different aspects, including the software implementation, realization, quality, and even economic value. Therefore, architects and designers need to combine different kinds of up-to-date views to address the various stakeholders' concerns and effectively support the evolution of the system.

Our focus is the construction of execution views, which we define as views that describe what the software embedded in a software-intensive system does at runtime and how it does it (Callo Arias, America and Avgeriou 2009b). The term runtime refers to the actual time that the software system is functioning during test or in the field. In contrast to module views, which describe how a system is constructed in terms of source code entities (e.g., modules, classes, functions, or methods), execution views describe how a system actually works 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. As we will describe later, using execution views, practitioners can get a global understanding of the actual runtime of a software system without being overwhelmed by the size and complexity of its implementation.

This chapter is an extension of our previous work, in which we presented how to analyze the runtime of a large software-intensive system (Callo Arias et al. 2008). The main contributions of the extension are:

- *An architecture reconstruction solution.* In our previous work we presented a dynamic analysis approach to extract and abstract up-to-date runtime information. Now, we present an improved approach, which 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, information to dig down into the details. The approach is an iterative and problem-driven architecture reconstruction process, which we implemented following a conceptual framework for architecture reconstruction using views and viewpoints (Koschke 2009, van Deursen et al. 2004).

- *Comprehensive description of the approach.* In our previous work we described how a metamodel and mapping rules supported the extraction and abstraction of runtime information. Now, we present an extended metamodel and details about the definition and use of mapping rules. In addition, we summarize the key elements of the viewpoints for execution views, which together with the metamodel represent explicit guidance for the (re)construction and use of execution view.
- *Case study.* In our previous work we presented the application of the dynamic analysis approach to enable the identification of dependencies between the runtime elements of the software in the Philips MRI scanner. We describe the validation of the architecture reconstruction solution with the construction of an execution profile view for the Phillips MRI scanner. According to the feedback from the practitioners involved in the validation, the execution profile view helped them to get an up-to-date global overview and insights about the actual runtime of key features of the Philips MRI scanner.

The organization of the rest of this chapter is as follows. In Section 3.2, we introduce our approach. Section 3.3 presents the metamodel and the execution viewpoints used by the approach. In Section 3.4, we present the dynamic analysis technique, including the source of runtime information and the use of mapping rules. In Section 3.5 and Section 3.6, we describe the application of the approach with the construction and use of an execution profile view for the Philips MRI scanner. Section 3.7 describes the technical contribution and potential limitations of the approach. Section 3.8 presents related work. Finally, in Section 3.9, we present some conclusions and future work.

3.2 Overview of the approach

As part of our research, we observed how architects and designers follow top-down analysis to support the incremental development and maintenance of the Philips MRI scanner. These practitioners start top-down analysis by constructing simple diagrams or sketches of what they consider the important parts of the system, or the architecture (Fowler 2003). The diagrams and sketches reflect the practitioners' mental models about the system, including its functionality, components, and how they interact with each other. The purpose of conducting top-down analysis using such representations is first to get a global understanding of the system and then, if it is needed for the problem at hand, to dig down for details.

We observed that practitioners prefer top-down analysis because they can focus on the problem at hand without being overwhelmed by the size and complexity of the

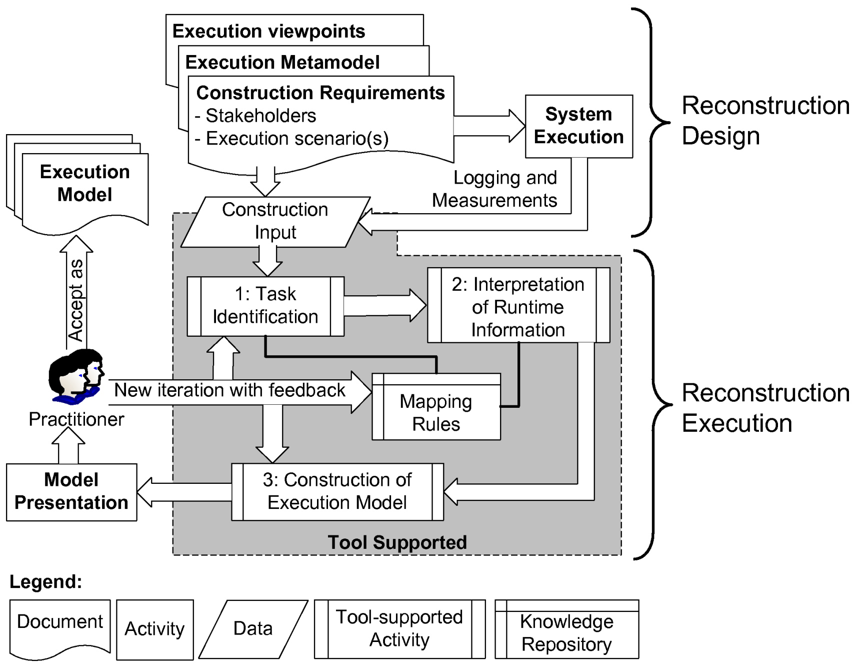


Figure 3.1: *Top-Down approach to construct execution views of a large software-intensive system*

system. Hence, we decided to develop a top-down approach that support practitioners, mainly architects and designers, in the construction of up-to-date execution views. The approach, illustrated in Figure 3.1, comprises: a set of elements that define the inputs for the approach, and a set of activities that enable the extraction-abstraction-presentation of runtime information. Together, these elements build an architecture reconstruction process based on Symphony (Koschke 2009, van Deursen et al. 2004), a conceptual process framework for architectural reconstruction. Symphony is an amalgamation of common patterns and best practices of reverse engineering using architectural views and viewpoints, which define architecture reconstruction as a process of two phases, reconstruction design and reconstruction execution.

3.2.1 Reconstruction design

The approach starts with a group of practitioners, e.g., an architect, responsible for documenting the architecture, and other personnel of the development organization who are domain experts with fair knowledge about the parts or features to be analyzed. With a set of execution viewpoints, described in Section 3.3.1, as guidelines, the architects identify the kind of concerns and problem(s) that can be addressed by constructing execution views, and then, if a match exists, select the kind of models to be constructed. For the reconstruction design, the experts contribute important domain knowledge, in order to identify:

- A set of representative execution scenarios that involve the part(s) or feature(s) of the system to be analyzed.
- A set of facts or domain knowledge that allows the decomposition of the system

runtime structure and behavior in terms of the entities and relationships organized by the execution metamodel described in Section 3.3.2. The facts can be collected in the form of text patterns, sketches, and references to existing documentation about the system or its runtime platform.

Then, runtime data, i.e. logging and measurements, can be collected running the system with the identified set of execution scenarios.

3.2.2 Reconstruction execution

With the construction input at hand, the approach continues through four activities. The first three are tool-supported activities that implement a dynamic analysis technique. The first and second activities, task identification and interpretation of runtime information, are for extracting and abstracting runtime information from the collected runtime data. Both activities rely on a repository of mapping rules, which is composed of regular expressions derived heuristically from the set of facts and domain knowledge identified in the design phase. In the third activity, construction of execution model, the architects select a subset of the extracted information and present it using the kind(s) of model(s) selected in the design phase. In the fourth activity, model presentation, the practitioners interested in the view analyze the constructed model and provide feedback about it. As Figure 3.1 illustrates, the result of the activity can determine one of the following situations:

- *Model acceptance:* The constructed models provide enough useful information to address the issue(s) in the development project. Therefore, the model is accepted as an actual execution model and becomes part of the execution view for the project at hand.

- *New iteration:* The model misses relevant information, hence an iteration of the reconstruction phase is needed. In the new iteration, the practitioners' feedback will be used to tune the extraction and abstraction of runtime information, including the extension or modification of the mapping rules repository and the subset of information in the constructed model.

In Section 3.5, we describe how several iterations are often necessary, especially when a model is constructed for the very first time. The next two sections describe in depth the elements and activities that support the design and execution phases of the approach.

3.3 Elements for the reconstruction design phase

Reconstruction design is considered 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 (Koschke 2009, van Deursen et al. 2004). In our approach, we support making explicit the viewpoints and metamodel that describe the conventions to construct, interpret, and use execution views. In the rest of this section, we summarize the viewpoints for execution views and describe an improved version of the metamodel presented in (Callo Arias et al. 2008).

3.3.1 Execution viewpoints

An architectural viewpoint frames particular concerns of the system's stakeholders and consists of the conventions for the construction, interpretation, and use of an architectural view (ISO/IEC 2007). The logical, process, development, and physical viewpoints (Kruchten 1995) are some of the existing viewpoints used to describe the architecture of software systems with multiple views. In practice, these viewpoints are customized, extended, or replaced according to the needs of the development organization and the characteristics of the system at hand. Often, the customization, extension, or replacement remains implicit in the head of the architect(s) that construct the views. Therefore, without the help of the original architect(s), other practitioners may not be able to (re)construct and use the same or similar views easily.

To facilitate the application of our approach and the use of the views constructed with it, we have defined and documented a set of execution viewpoints (Callo Arias, Avgeriou and America 2009, Callo Arias, America and Avgeriou 2009b). The definition includes the customization of some viewpoints from the literature, our observations from describing the runtime of the Philips MRI scanner, and the preferences of key practitioners, which we collected conducting dedicated interviews. The documentation includes the specification of concerns and conventions that guide the construction and use of the following three kinds of execution views:

i. *Execution profile view:* The models in an execution profile can be used to provide overviews and facilitate the description of details about the runtime realization of a given system feature, without being overwhelmed by the size and complexity of the system implementation. The case study, presented in Section 3.5, focuses on the construction of a view of this kind. Some of the questions that can be addressed with this kind of view are:

   - What are the major components that realize a given system feature?
   - What are the major tasks that build the actual runtime workflow of key features?
   - What are the dependencies between runtime elements?
   - What is the development team that develops or maintains a given system's function?

ii. *Execution concurrency view:* The models in an execution concurrency view can be used to provide overviews of how the runtime elements of a software-intensive system execute concurrently at different levels of abstraction. An example of this kind of view is presented in (Callo Arias, Avgeriou and America 2009). Some of the questions that can be addressed with this kind of view are:

   - Which runtime elements execute concurrently?

- How does the runtime concurrency match the designed concurrency?
- What are the aspects that constrain or control the system's runtime concurrency?
- What are the bottlenecks and delays of the system and their root causes?
- What are the opportunities to improve the concurrency of the system?

iii. *Resource usage view:* Resource usage models can be used to provide overviews and facilitate the description of details about how the elements, e.g. component and process, of a system use hardware resources at runtime. An example of this kind of view and how to construct it is presented in (Callo Arias, America and Avgeriou 2009a). Some of the questions that can be addressed with this kind of view are:

- How to assure adequate resource usage and justify the development effort needed to accommodate hardware resources changes?
- What are the metrics, rules, protocols, and budgets that rule the use of resources at runtime?
- How do software components and their respective processes consume processor time or memory when running key execution scenarios?
- Does the realization of the system implementation have an efficient resource usage?

The questions listed above reflect some of the stakeholders' concerns with respect to the runtime of a software system. The separation or classification of these concerns into three different kind of execution views is important to facilitate the construction, analysis, and communication of models with information that address each set of concerns separately. It is not possible to construct a single, usable execution view to address all of these concerns. Table 3.1 is a summary of other key aspects, documented by the execution viewpoints, which supports the construction and use of execution views as follows:

- *Stakeholders:* The kind of practitioners concerned about the runtime of the system, who need or can contribute in the construction, analysis, and communication of execution views.

- *Development activities:* Some of the usual activities in a development project where practitioners need to describe or analyze the actual runtime of a system.

- *Runtime entities:* Subsets of entities, elements in the execution metamodel described in Section 3.3.2, that determine the abstraction level to described the runtime of system used in each kind of model.

- *Model:* The kind of representations that can be constructed with the approach to organize and present runtime information. The documentation of the corresponding execution viewpoints includes details about the notations and representations in each kind of model. Therefore, given the kind of execution view to be constructed, the practitioner can select the kind of model that presents and organizes best the information that they need.

Overall, the execution viewpoints support the elicitation of the problem that requires the construction of up-to-date execution views. This includes the identification of the concerns to be addressed, the stakeholders to be involved, and how to represent the required information. Further details about the elements that build the execution viewpoints can be found in (Callo Arias, Avgeriou and America 2009, Callo Arias, America and Avgeriou 2009b), which are cited by the new version of the ISO/IEC CD1 42010 standard as a representative example of viewpoints and how to document them.

Table 3.1: *Summary of the elements in the viewpoints for 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.		
View	**Models**	**Runtime entities**
Execution Profile	Functional mapping	Task, component, process, and data and code resources
	Execution workflow	Processing node and task
	Matrix model	Task, component, and quantifications
	Sequence diagrams	Task, component, process, and data and code resources
Execution Concurrency	Workflow concurrency	Processing node, task, and component
	Process and thread structure	Component, process, and thread
	Control and data flow	Process, thread, and control and data interactions
Resource Usage	Task resource usage	Task and hardware usage overtime
	Component resource usage	Component and hardware usage overtime
	Thread resource usage	Thread and hardware usage overtime

3.3.2 Execution metamodel

Practitioners are often familiar with abstractions to describe the structural decomposition of a system, but less familiar with abstractions that allow them to describe the

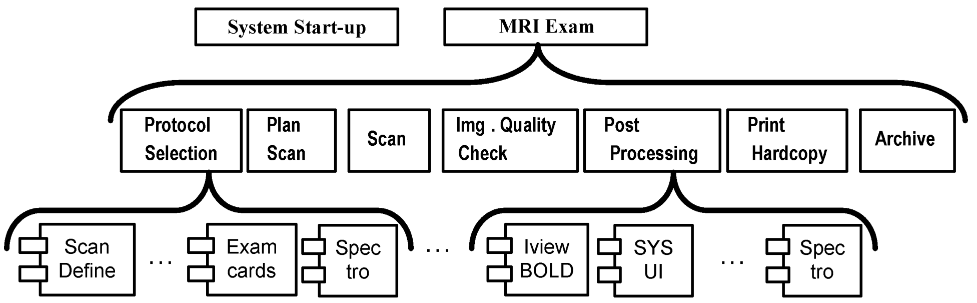


Figure 3.2: *A high-level description of the runtime of the Philips MRI scanner*

runtime of a software-intensive system. Figure 3.2 shows a description of two key runtime features of the Philips MRI scanner in the field. This description shows at a high-level the steps in the execution scenario of one of the features and the software components that implement each step. The usual approach to extend this description in a top-down fashion would be to map software components to source code artifacts such as code packages, modules, classes, or methods. Using these abstractions can be overwhelming for systems with large and heterogeneous implementations. In addition, one can easily miss other relevant elements, such as data and hardware, and relationships such as dynamic communication links that play an important role in the runtime of a software-intensive system.

The metamodel illustrated in Figure 3.3 organizes a set of concepts and relationships between them, which play a role in the runtime of a software-intensive system. 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 belongs 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 architectural concepts, elements of the runtime platform, and runtime resources. Instead, we consider it an useful subset that matches the characteristics of the Philips MRI scanner and similar software-intensive systems.

Execution scenarios.

A scenario is a brief narrative of expected or anticipated use of a system from both development and end-user viewpoints (Kazman et al. 1996). Scenarios can be described 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 runtime of a system. 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.

Tasks.

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 3.2 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 start-up involves a wide range of interactions that the development organization wishes to control and analyze, thus it also 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.

Processing nodes.

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 elements of a software-intensive system are deployed and run.

Software components and processes.

In our approach, we consider a software component as a set of processes that belong together. A process is an entity handled by the operating system or runtime platform hosting the software of a software-intensive system. A process represents a running application, including its allocated resources: a collection of virtual memory space, code, data, and platform resources. Large software systems are often composed of many processes or running applications. The metamodel describes that one or more running processes make up a software component, because it is possible to group them taking into account two types of relationships between them:

- Actual parent-child relationships can be established between processes when a main process (parent) creates another process (child) to delegate a temporary or specific function.

- Design relationships are established by the development organization to distinguish that a set of processes belong together because they share functional or non-functional characteristics. This kind of distinction is used to reduce complexity and facilitate the analysis of the system.

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.

Interactions between components.

Interactions between individual processes can be identified and therefore between their corresponding software components. Based on our observations and the literature (Kruchten 1995, Rozanski and Woods 2005), some interesting runtime interactions at the architecture level are:

- Data-sharing interactions allow two or more processes to share and access one or more data structures concurrently. Examples include shared memory, databases, and file storages.
- Procedure call interactions are some sort of inter-process function calls, often based on remote procedure calls or message-passing operations.
- Execution coordination interactions allow two or more processes (or threads) to signal to each other when certain events occur, e.g., access to shared data. Semaphores and mutexes are common coordination mechanisms at the process or thread level.

Instances of these types of interactions between software components can be identified analyzing runtime activity of their processes. For instance, analyzing read and write 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.

Threads and runtime 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 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 activity represents the loading and execution 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 activity represents the utilization of platform resources. The processing nodes and the runtime platform of a software-intensive system provide

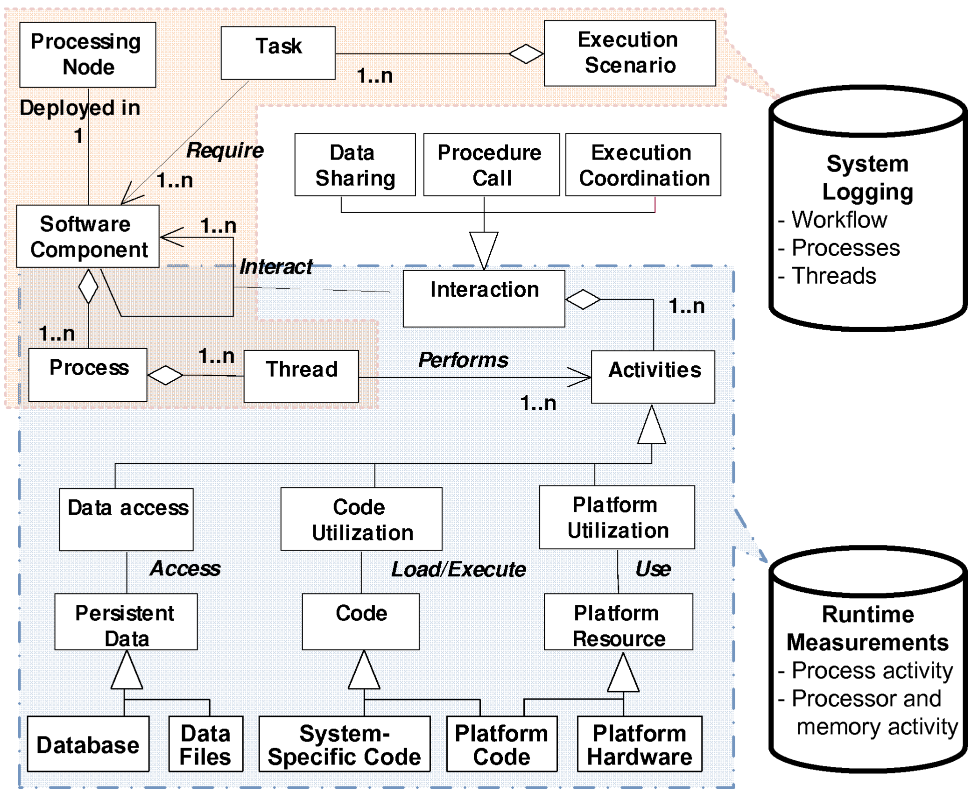


Figure 3.3: *A metamodel of the runtime of a software-intensive system and the sources of information in terms of the elements in the metamodel.*

hardware and software resources that the software of a software-intensive system uses or controls 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.

3.4 The reconstruction execution phase

The reconstruction execution is defined as an extract-abstract-present process, which yields the architectural view needed to address the problem that triggered the construction activity (Koschke 2009, van Deursen et al. 2004). In this section, we describe the source of information and the dynamic analysis technique to extract-abstract-present runtime information in our approach.

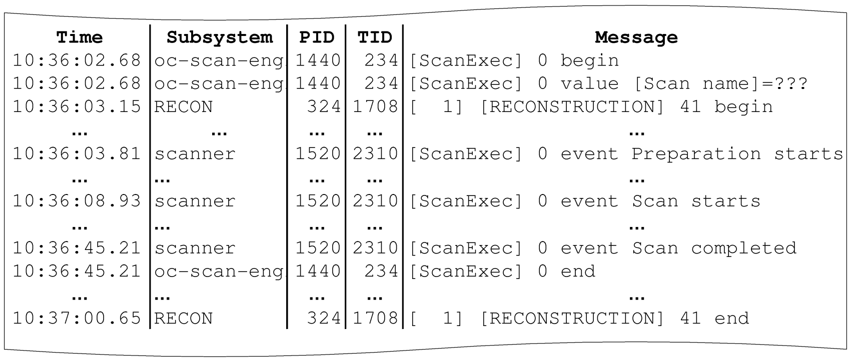
3.4.1 Sources of runtime information

Figure 3.3 illustrates that runtime information, in terms of the abstractions or concepts described in the execution metamodel, can be extracted and abstracted from sources like logging and runtime measurements.

- *Logging:* Most large software systems use logging mechanisms to record and store information of their specific activities into log files. For example, Figure 3.4 shows a piece of a log file, which records the runtime events in the workflow of the functionality of an MRI scanner. The '...' in the figure means that many other kinds of logging messages were filtered out for the example. In practice, logging messages are recorded with mechanisms that are implemented according to the implementation technology of the system and the preference of the development organization (Moe and Carr 2001). Regardless of the logging mechanisms, it is common that developers, testers, and other specialized users use logging information for understanding, debugging, testing, and corrective maintenance (Moe and Carr 2001, Yantzi and Andrews 2007, Jiang, Hassan and Hamann 2008). For our approach, logging is a source of workflow information, like the messages illustrated in Figure 3.4, which we used to identify high-level abstractions such as the tasks and software components in a execution scenario (defined in Section 3.3.2). Analyzing logging is not a trivial activity, especially when the development teams adopt different logging formats, naming conventions, and even different logging mechanisms. Rule-based, codebook-based, and AI-based mechanisms are available in the literature to support the abstraction and analysis of logging (Jiang, Hassan and Hamann 2008). As we will describe in Section 3.4.2, our approach is based on rule-based mechanism, which relies on a set of hard-coded rules that map log lines to runtime abstractions.

- *Runtime measurements:* Most runtime platforms offer tools and mechanisms to collect runtime measurements. Runtime measurements record the activities of processes and resources such as processors and memory in a system's processing node(s). For example, process activity and resource activity can be monitored with tools like Process Monitor in the Microsoft Windows platform (Microsoft Corporation 2010b). Other tools are also available to monitor process activity in the Linux and Unix platforms (Gavin 1998). Runtime measurements are available in semi-standardized formats independently of the implementation technologies. Runtime measurements 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). For our approach, runtime measurements provides information about process activity, which is used to identify elements such as processes, threads, and their activities on resources like data, code, and hardware devices (defined in Sections 3.3.2).

In addition to the information provided by logging and runtime measurements, these sources can be easily collected and combine for most software systems without considerable overhead. Logging mechanisms are available in most systems and the overhead produced by these mechanisms is part of the normal system behavior or within the expectations of the development organization. Having logging and runtime measurement tools running at the same time eases the collection and synchronization of runtime data. Figure 3.5 illustrates the latter. The figure shows that when a log message is recorded (written) in a log file, a tool monitoring process activity can capture the write event that happens on the log file. The data captured for the write event can



```
Time        Subsystem    PID  TID   Message
10:36:02.68 oc-scan-eng 1440   234 [ScanExec] 0 begin
10:36:02.68 oc-scan-eng 1440   234 [ScanExec] 0 value [Scan name]=???
10:36:03.15 RECON        324  1708 [  1] [RECONSTRUCTION] 41 begin
    ...         ...      ...   ...                ...
10:36:03.81 scanner     1520  2310 [ScanExec] 0 event Preparation starts
    ...     ...          ...   ...                ...
10:36:08.93 scanner     1520  2310 [ScanExec] 0 event Scan starts
    ...     ...          ...   ...                ...
10:36:45.21 scanner     1520  2310 [ScanExec] 0 event Scan completed
10:36:45.21 oc-scan-eng 1440   234 [ScanExec] 0 end
    ...     ...          ...   ...                ...
10:37:00.65 RECON        324  1708 [  1] [RECONSTRUCTION] 41 end
```

Figure 3.4: *Example of a log file with workflow messages*

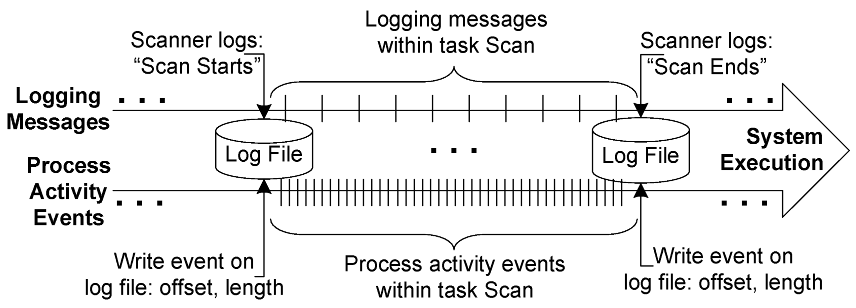

Scanner logs: "Scan Starts"
Logging messages within task Scan
Scanner logs: "Scan Ends"
Logging Messages
Log File
Log File
System Execution
Process Activity Events
Write event on log file: offset, length
Process activity events within task Scan
Write event on log file: offset, length

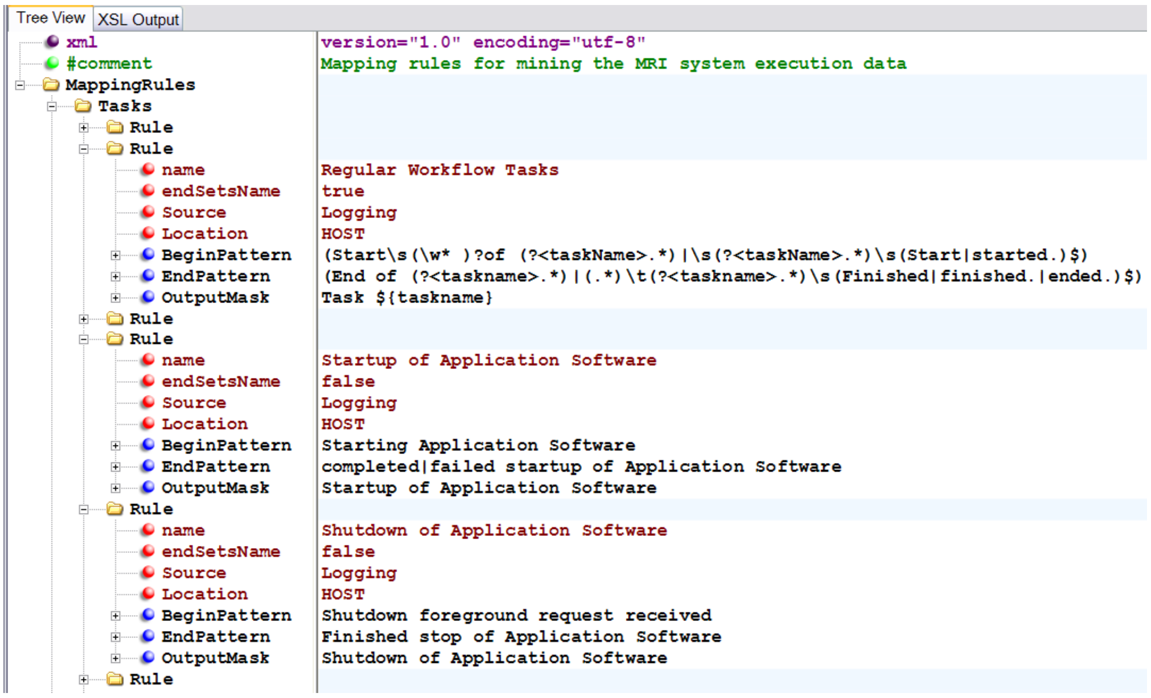
Figure 3.5: *Simultaneous gathering of runtime data from logging and process activity*

include information about the acting process, the size of the message, and the timestamp of the write event. In our approach, we exploit this situation to synchronize the logging and other runtime measurements.

3.4.2 Mapping rules repository

As illustrated in Figure 3.1, the first two activities of the dynamic analysis use a mapping rules repository. Mapping rules can be both formal and informal specifications that describe how entities in a given level of abstraction can be mapped to entities in a higher level of abstraction. In the case of our approach, mapping rules specify how to map data from logging or runtime measurements to instances of the elements and relationships organized by the execution metamodel, described in Section 3.3.2. The mapping rules in our approach are mainly regular expressions derived from the set of facts and domain knowledge identified in the design phase.

Figure 3.6 and Figure 3.7 show the structure of the mapping rules repository and the definition of some mapping rules. The repository is an XML file and its main entries classify mapping rules based on the kind of runtime entities, e.g. tasks, software components, and resources. As the figures show, the definition of a mapping rule captures a set of parameters according to the kind of runtime entity. Among the kind of definitions, regular expressions are used to specify text patterns (see BeginPattern, EndPattern, or processDescriptionPattern in the figures). These different parameters are described in the next subsections. Defining mapping rules and populating the



```
xml                     version="1.0" encoding="utf-8"
#comment                Mapping rules for mining the MRI system execution data
MappingRules
  Tasks
    Rule
    Rule
      name              Regular Workflow Tasks
      endSetsName       true
      Source            Logging
      Location          HOST
      BeginPattern      (Start\s(\w* )?of (?<taskName>.*)|\s(?<taskName>.*)\s(Start|started.)$)
      EndPattern        (End of (?<taskname>.*)|(.*)\t(?<taskname>.*)\s(Finished|finished.|ended.)$)
      OutputMask        Task ${taskname}
    Rule
    Rule
      name              Startup of Application Software
      endSetsName       false
      Source            Logging
      Location          HOST
      BeginPattern      Starting Application Software
      EndPattern        completed|failed startup of Application Software
      OutputMask        Startup of Application Software
    Rule
      name              Shutdown of Application Software
      endSetsName       false
      Source            Logging
      Location          HOST
      BeginPattern      Shutdown foreground request received
      EndPattern        Finished stop of Application Software
      OutputMask        Shutdown of Application Software
    Rule
```

Figure 3.6: *Examples of mapping rules for task identification*

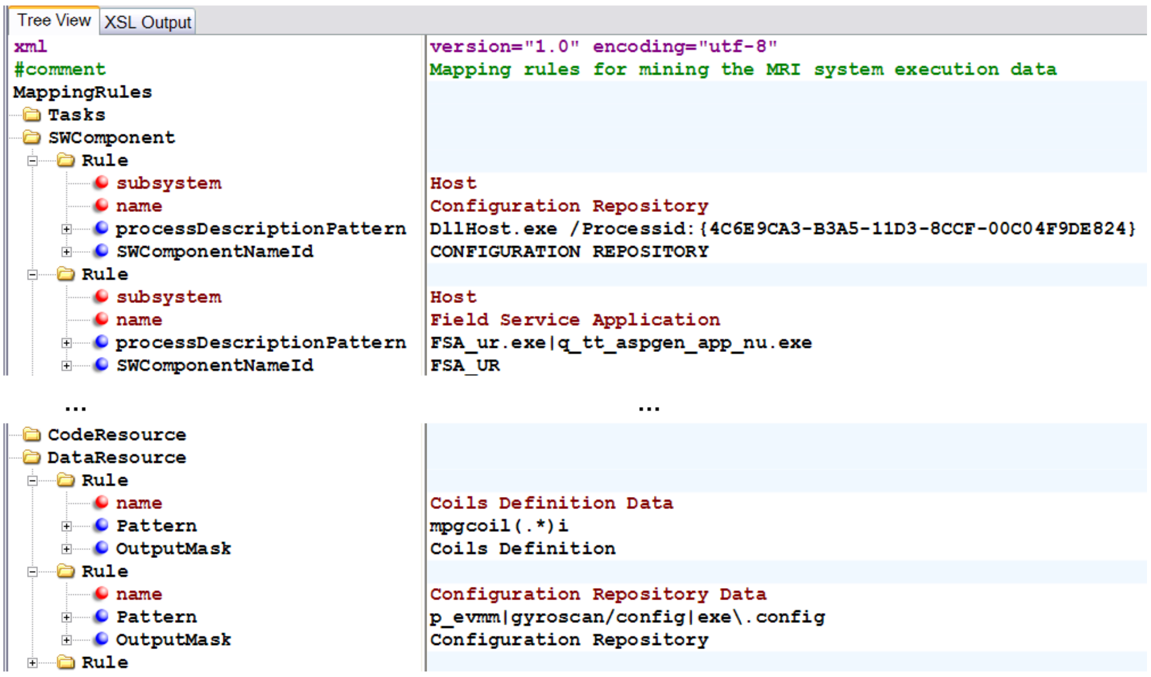
repository is a problem-driven and iterative process. At first, mapping rules are defined only to extract runtime information of interest for a problem at hand. Later, in a new iteration or a new reconstruction activity, we can reuse some mapping rules or redefine them, either to zoom in on details or aggregate them. The next sections describe the use of mapping rules for task identification and interpretation of runtime information.

3.4.3 Task identification

The task identification activity aims at: 1) the identification of the tasks that build the workflow of an execution scenario, and 2) the synchronization of logging and runtime measurements for each identified task. Figure 3.8 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, for the execution scenario. The output is a set of task data structures. A task data structure 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 activity is implemented as follows:

- *Find logging events:* Assuming that the collected runtime measurement contains monitored process activity that captures write events in the log fie, as illustrated in Figure 3.5, the runtime measurements are sequentially parsed to identify write events in the log file.

- *Extract logging messages:* When a write even is identified, the parameters in the event



```
xml                                   version="1.0" encoding="utf-8"
#comment                              Mapping rules for mining the MRI system execution data
MappingRules
  Tasks
  SWComponent
    Rule
      subsystem                       Host
      name                            Configuration Repository
      processDescriptionPattern       DllHost.exe /Processid:{4C6E9CA3-B3A5-11D3-8CCF-00C04F9DE824}
      SWComponentNameId               CONFIGURATION REPOSITORY
    Rule
      subsystem                       Host
      name                            Field Service Application
      processDescriptionPattern       FSA_ur.exe|q_tt_aspgen_app_nu.exe
      SWComponentNameId               FSA_UR
  ...                                 ...
  CodeResource
  DataResource
    Rule
      name                            Coils Definition Data
      Pattern                         mpgcoil(.*)i
      OutputMask                      Coils Definition
    Rule
      name                            Configuration Repository Data
      Pattern                         p_evmm|gyroscan/config|exe\.config
      OutputMask                      Configuration Repository
    Rule
```

Figure 3.7: *Examples of mapping rules for the interpretation of runtime information*

information (offset and length) are used to extract the corresponding text message(s) from the log file.

- *Apply mapping rules:* The mapping rules for task identification in the repository are applied to the extracted logging message. The main and first mapping to be applied is ”Regular Workflow Tasks”, shown in Figure 3.6. This rule represents our assumption that the tasks in an execution scenario are often delimited by logging messages that contain variations of start or end like text patterns, like the messages shown in Figure 3.4. Other mapping rules with specific patterns like ”Startup of Application software”, shown in Figure 3.6, are defined and applied according to the problem at hand and the knowledge captured in the design phase. The application of a mapping rule can lead to one of these situations:

  - *Begin task*. When the text of a logging message matches the BegingPattern of a mapping rule, a task data structure is created including the corresponding kind of task, name, and the identification of the process that logged the message.

  - *End task*. When the text of a logging message matches the EndPattern of a mapping rule and a task structure exist with similar parameters, the task structure is closed and stored updating its parameters. Attributes of the mapping rule, like endSetName, may indicate that the EndPattern provides the final name or identifier for the task.

- *Runtime activity in task:* When the logging message does not match a mapping rule or the parsed runtime measurement is not a logging event, the data item is considered as source of runtime information and it is sequentially added to the last task

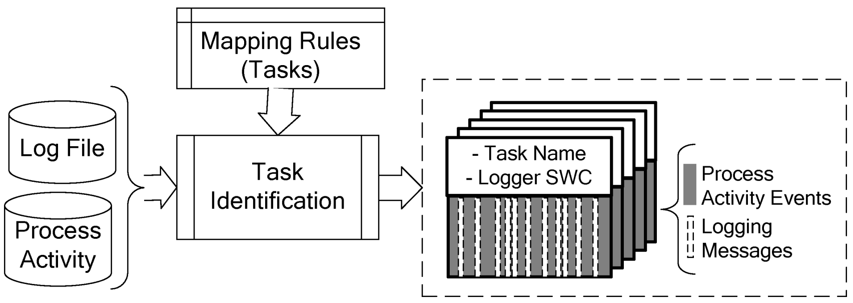


Figure 3.8: *Identifying task information*

structure being created. When multiple task structures are open because they run concurrently, the data item is added to the task structure with similar processing node, or logging process, or thread id.

The tool support for this activity is a set of Python scripts that we wrote to parse logging files and runtime measurements files. Both kinds of files are parsed as comma-separated values files taking into account their original specification, e.g., column names and data type. For situations where mapping rules like "Regular Workflow Tasks" cannot be applied or facts and domain knowledge about the system are not available to define specific mapping rules, the analysis of timestamps is an alternative to split execution scenarios into tasks and synchronize logging and runtime measurements.

The task identification activity plays an important role in the top-down approach. On the one hand, it helps to split a large amount of runtime data into manageable data sets, i.e. task data structures. These data structures can be stored independently and later processed on demand or in parallel. On the other hand, the output of this activity provides relevant high-level information. The information includes the workflow messages that represent the boundaries of an identified tasks and a set of process that can be considered the root of software components. The latter is based on the assumption that only important or main processes write log messages. In cases where an overview is needed before going into details, this information can be used already to construct high-level descriptions, e.g. execution workflow models like the one in Figure 3.15.

3.4.4 Interpretation of runtime information

The activity of interpreting runtime information, extends the runtime information extracted in the task identification activity according to the kind of view to be constructed and the problem to be addressed. The extension constitutes the interpretation of the runtime data bundled in the task data structures. Figure 3.9 illustrates the concept behind the interpretation using mapping rules. According to the sources of information, mapping rules can be created to abstract logging messages (LMR) and runtime measurements, i.e., process activity events (PAMR). In all cases, system- and platform-specific facts are used to define mapping rules. As the figure illustrates LMR and PAMR are partial functions because they do not necessarily map every logging message (L_i) or every process activity event (A_j) into runtime interactions according to the elements and relationships in the execution metamodel. The application of mapping rules for the interpretation of runtime information is a combination of LMR and PAMR, text patterns are considered in the text of logging messages and in the text of runtime measurements at the same time. This combination is need to generate one or more of the elements in an execution interaction tuple. The elements in such a tuple are:

- A *subject* represents a software component or a process entity, which can be identified applying "SWComponent" mapping rules, shown in Figure 3.7. The definition of this kind of mapping rule can capture: facts about the runtime platform to identify important processes as software components (see "Configuration Repository" rule), or system-specific design relationships to group individual processes into a software component (see "Field Service Application" rule).
- A *verb* represents an execution activity (e.g. read, write, load, and execute), which is often part of the text in the logging message or process activity event.
- An *object* represents data, code, platform resources, or other software component and process. Data, code, and platform resources can be identified applying "CodeResources" or "DataResources" mapping rules, shown in Figure 3.7. The definition of these kinds of mapping rules capture facts about the system and the platform file structure.
- *info* is an alternative element that can contain information such as the thread identifier or a timestamp, which can be used to zoom in on details.

Figure 3.10 illustrates the interpretation of runtime information activity for a task in a scenario. The inputs are the task data structure and the mapping rules repository, in particular the kind of mapping rules shown in Figure 3.7. These mapping rules are applied to the logging messages and process activity events bundled in the task data structure to abstract them into execution interactions in terms of the elements organized by the execution metamodel. The output of the activity is a set of interaction tuples, which we store as a graph structure called interaction graph, shown at the right side of the figure. As an example, the interpretation of a process activity event that describes a running Process B creating another process B1 within its thread T, generates the tuple: (Process B, Create, Process B1, Thread T). Subsequently SWComponent mapping rules, defined with logging patterns, can be applied to map Process B and Process B1 to the corresponding software component(s). Similarly, CodeResource or DataResource mapping rules can be applied to the object in the interaction tuple to identify data and code aggregations.

An additional output of interpretation of runtime information is a list of the identified software components and the aggregation or system resources within the task. The tool support for this activity is a set of Python scripts that we wrote to parse the text of logging messages and runtime measurements, i.e., monitored process activity.

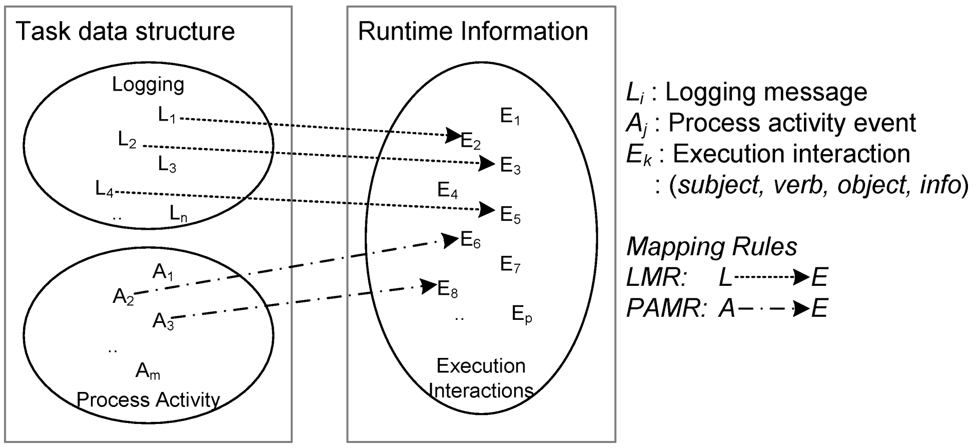


Figure 3.9: *Concepts for the interpretation of runtime information*

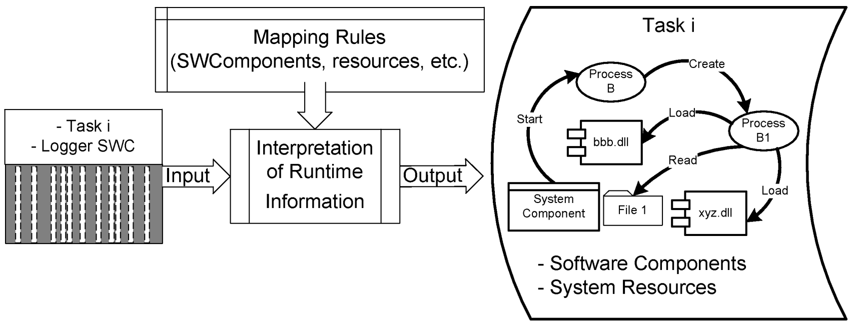


Figure 3.10: *Overview of the interpretation of runtime information*

To build and store interaction graphs, the scripts use the NetworkX library (*NetworkX* n.d.). The implementation and use of graph structures facilitate the analysis and query of the extracted runtime information in the next activity, construction of execution model.

3.4.5 Construction of execution model

This activity focuses on the construction of the execution model(s) that will present the runtime information that address the concerns or questions identified in the reconstruction design phase. As we described in Section 3.3.1, the execution viewpoints include a set of specific questions, the kinds of models that can be constructed to address the questions, and the runtime elements that are used in each kind of model. With this specification as a reference, the realization of this activity is a top-down operation that consist of the following steps:

- The corresponding tool, a .Net application, displays the set of high-level elements, e.g. tasks, software components, and data or code aggregations, that were automatically identified in the data stored in the interaction graphs of the execution scenario under analysis.

- With the question or concern in mind, we select a subset of the high-level elements and the kind of model to be constructed.
- With the subset of elements as input, the tool queries the interaction graphs' structure for the actual runtime entities, that build or are part of the high-level elements, and their respective runtime activity.
- According to the kind of model, the tool can automatically generate representations like the one in Figure 3.12, the matrices in Figure 3.16, or data inputs for other visualization tools.

The .Net application that support the construction model activity includes a user interface to run the Python scripts that implement the previous two activities, task identification and interpretation of runtime information. As we stated before, another feature of the tool is the generation of data inputs for some visualization tools. These tools include Graphviz (*Graphviz - Graph Visualization Software* n.d.), to present models like the one in Figure 3.12, UML Graph (*Automated Drawing of UML Diagrams* n.d.) to generate the sequence diagrams in Figure 3.14, and Microsoft Excel to generate models like the one in Figure 3.15. In the next section, we describe more about the iterative construction of these various models and their respective use and value.

3.5 Validation of the approach

The approach has been validated across several development projects of the Philips MRI scanner (see Section 1.2.1). In our previous work (Callo Arias et al. 2008), we reported the results of the approach for dependency analysis. In this chapter, we describe the validation of the approach with the construction of a set of execution models that build an execution profile view for the Philips MRI scanner. In the rest of this section, we summarize the phases for the construction of the execution profile view. In Section 3.6, we describe some use cases for the view that we observed during the validation.

3.5.1 Design reconstruction for an execution profile view

As we described in Section 3.3.1, an execution profile view provides overviews and details about the actual runtime structure and behavior of a system. In this case, the main motivation was to get up-to-date overviews and insights about the runtime of key features of the Philips MRI scanner. For this purpose, the development organization selected three key features: 1) the management of MRI coils (system-specific hardware devices), 2) the standard clinical scan procedure, and 3) the startup process of the software system. These features were selected for three reasons. First, knowledge about each feature was required for the planning of several development projects. Second, the runtime of these features change often, not only due to changes in the software elements but also in the hardware elements that compose the system.

Third, these features are tightly coupled with performance and distribution requirements; this implies that tuning any of the features to match changing requirements will influence the runtime structure and behavior of the system. The characteristics of the reconstruction design in our case are as follows:

- *Stakeholders:* For each feature, an architect and a designer were assigned as the main stakeholders. The architect was responsible for documenting and communicating the constructed models. The designer was identified as the feature or domain expert. In addition, a platform supporter and some developers were involved based on their interest in the design and implementation of a solution for the problem being addressed by the development project.

- *Scenarios:* The feature expert selected some of the usual system test cases as representative execution scenarios for the selected features. For the validation, we only focus on "clean" scenarios, e.g., scenarios without faults.

- *Facts or domain knowledge:* Sketches made by the experts were the main forms to capture facts and domain knowledge about the selected features. In the sketches, the experts depicted the system elements and their relationships, which were supposed to play a relevant role in the runtime of the given feature. In addition, we used some system documents to understand the logging mechanism and the structure of the logging file used by the system.

- *Runtime data:* We collected the runtime data by running the selected execution scenarios with the assistance of system operators. These practitioners were necessary to setup and operate the system according to the specifications of the execution scenario. Table 3.2 is a summary of the logging and runtime measurement that we collected for the construction of the models in the view. To collect runtime measurements, we used the Process Monitor Tool (Microsoft Corporation 2010b) since the runtime platform of the system is Microsoft Windows. The summary also includes the technical information or documentation that we studied to identify the text patterns or facts that we used to define the mapping rules for the execution profile view of the Philips MRI scanner. The size of the data is not included because, as we described in Section 3.4.4, our mapping rules are partial functions and we do not have an accurate measurement on how many logging message or process activity events were actually used to extract high-level information.

3.5.2 Execution reconstruction for an execution profile view

The models that we constructed as part of the execution profile view for the Philips MRI scanner are functional mapping, execution workflow overview, matrices, and sequence diagrams models. Some simplified examples of these kind of models are illustrated in Figures 3.12, 3.15, 3.16, 3.13, and 3.14. To produce such useful models, we went through several iterations of the execution construction phase to cope with three main situations:

Table 3.2: *Summary of runtime data for the construction of an execution view.*

Data	Extracted information	Source of patterns
Logging:		
Workflow messages	Tasks, software components, processes, and threads	MRI Log guidelines
Debug messages	Major code modules	
Runtime measurements:		
File access events	HW and SW configuration-setting data	MRI naming conventions
	System database and data elements DLLs, assemblies, and dynamic wrappers Script programs	Filesystem structure and (Microsoft Corporation 2010b)
Process activity	Software components, processes, and threads	(Microsoft Corporation 2010b, *About Processes and Threads (Windows)* n.d.)
Windows registry access	COM elements, HW and SW configuration-setting data, and Communication services and platform resources	(*About Processes and Threads (Windows)* n.d., *Registry (Windows)* n.d.)

i. *Tune the task structure of a scenario:* For every feature, we started presenting the task structure found by applying the "Regular Workflow Tasks" mapping rule, shown in Figure 3.6. At first, it exposed an approximation of the actual task structure of a scenario, but additional iterations were necessary to decompose coarse-grained tasks, aggregate repetitive occurrences of a task, correct gaps between tasks, or even define a different task structure. For most of these cases, the iterations included the definition of mapping rules with specific logging messages, e.g. the "Startup of Application Software" rule shown in Figure 3.6.

ii. *Classify and aggregate runtime elements:* An initial set of mapping rules based on patterns about naming conventions and the filesystem structure enabled the classification of software, data, and even hardware elements, in a execution scenario, into high-level aggregations, e.g. some files were classified as data files, system configuration files, and so on. Yet, it was not enough to address specific problems. For example, we run an iteration to distinguish between the runtime activity on the configuration data for the MRI Coils devices, from the runtime activity on the system configuration data. This iteration included the definition of the "Coils Definition Data" rule, shown in Figure 3.7, which enabled the classification and aggregation of hundreds of data files and runtime activity on them. Similar iterations were necessary for the other features and different kinds of runtime elements and activity. For example, Figure 3.11 illustrates three system specific classifications, based on the elements in the metamodel, for which we had to define specific mapping rules.

iii. *Filter out irrelevant runtime information:* The stakeholders' feedback can indicate that part of the recovered runtime information does not belong or is exclusive to the realization of the feature under analysis, or simply is not relevant for the problem at

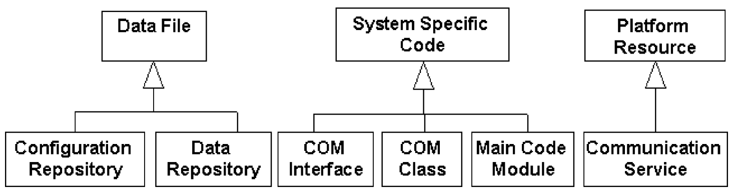


Figure 3.11: *Further specialization or classification or runtime elements*

hand. Therefore, additional iterations are necessary to identify and filter out such runtime information from a new execution model. The iterations entail running the interpretation of runtime information activity with fewer mapping rules or simply editing the constructed model. For example, the first time that we constructed functional mapping models like the one in Figures 3.12, part of the information in the model was about the runtime activity of the antivirus application and other system utilities running in the computers of the system. However, the expert decided that this information was not relevant for the problem at hand, so we run an iteration to filter out such information from the constructed model.

Most of the iterations included the gradual definition of mapping rules and the tuning of the diagrammatic representations. Basic mapping rules like those based on workflow messages and the file system structure were defined upfront. The definitions of other special rules to create specific aggregations or tasks were only possible and necessary when the problem was better understood. Overall, the mapping rules recorded a number of facts about the system that were implicit in the text of logging messages, runtime measurements, technical documents, and in the mental models of some practitioners. Since we stored the definition of the mapping rules in a repository, reconstructing the same models after the first time, e.g., to verify changes or conformance, did not include more iterations than the ones to filter out irrelevant information.

3.6 Use cases for an execution profile view

Software-intensive systems like the Philips MRI scanner are evolved or incrementally developed. This implies that a development cycle focuses on changing parts of the system rather than the system as a whole. Therefore, before, during, and after changes are performed, a proper understanding of the design, implementation, and realization of the parts to be changed is required. To achieve such understanding, practitioners perform a number of analysis activities. In this section, we describe the analysis activities that we supported by the construction of the execution profile view.

3.6.1 Feature analysis.

We observed that during the incremental development of the Philips MRI scanner, feature analysis is necessary to plan and perform changes on existing features. This is especially the case when it is not obvious which software elements are involved in the realization of a changing feature. Consequently, it is not possible to have an accurate estimation of the part that will be touched, or the development effort and resources that are required to perform the change. Typically, feature analysis is done to identify those parts of the source code which implement a specific set of related features (Eisenbarth et al. 2003). However, feature analysis at the granularity of source code elements such as functions, methods, or classes is impractical for large systems like the Philips MRI scanner. Instead, we observed, that higher-level abstraction and ways to breakdown complex features into less complex units are more appealing for practitioners, e.g., architects and designers, steering the development of large and complex software-intensive systems.

Functional mapping models constitute the main contribution of the execution profile view to support feature analysis. A functional mapping model is a graph-based representation of traceability links that exist in the realization of a system feature based on an execution scenario. Figure 3.12 shows and example of a functional mapping model and its notations. The model in the figure is a simplified version of a model that we constructed to analyze the system feature for the management of MRI coil devices. The model helped us to describe the main tasks (remove, add, and enable coil) that compose the feature, the relations between the tasks and the system software components, the runtime processes that belong together and build up the software components, and the runtime activity performed by the respective processes to use resources such as code modules and data elements.

Using functional mapping models, the practitioners involved in the validation were able to analyze a number of aspects for planning changes. For example, the links between tasks and components in Figure 3.12 shows that to change the way coils are removed (Remove Coil task), the participation of the teams developing the COILCONFIG UPDATE and CONFIGURATION REPOSITORY components will be required. Similarly, the description of the runtime activity of the processes, in the involved software components, shows that the change will also require knowledge about the drivers (Coil Code modules) and the configuration data (both Coil Definition and Configuration Repository).

In practice, a development project involves more that one system feature and each feature can be represented by more than one execution scenario. Therefore, producing useful functional mapping models requires the input from experts, especially to carefully select the features to be analyzed and their respective set of representative execution scenarios.

3.6.2 Dependency analysis.

During the incremental development of a large system like the Philips MRI scanner, dependencies imply ramifications of changes between parts of the system or features that are developed by different teams of the organization. Therefore, without explicit and up-to-date information about dependencies, it is not possible to have an accurate view on how changes in a given feature can propagate to other parts of the system.

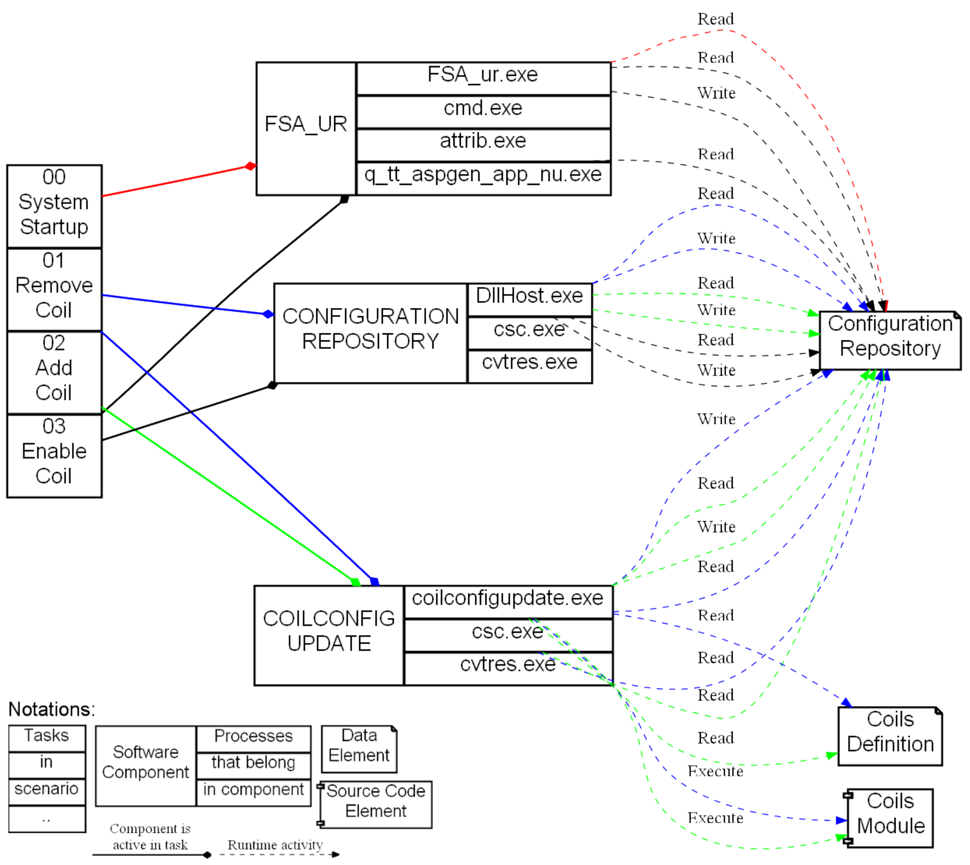


Figure 3.12: *Example of a functional mapping model for the Philips MRI scanner*

Typically, source code constructs such as function calls, shared variables, and shared libraries are characterized as dependencies. However, in a large and complex system like the Philips MRI scanner, relationships that determine dependencies are not always established in the source code, but can occur in many different ways at runtime, e.g. inter-processes communication, shared data, and temporal relationships.

Matrix models like the ones in Figure 3.13 and Figure 3.16 provide details that complement high-level information such as the information given in functional mapping models. Table 3.3 lists the types of matrices that we constructed for the execution profile view. The tool support for our approach provides filtering facilities to choose between these types and the runtime elements than can be tabulated and quantified in the cells of a matrix. Figure 3.13 show two matrices, (b) and (c), of type III that provide details to determine if high-level relationships, described by the functional mapping model (a), constitute dependencies between software components and data repositories. The model (a) in the figure is a simplified version of a functional mapping model for the clinical scan procedure performed with the Philips MRI scanner. The matrix (b) is used to find the data elements in the UI MENU STRUCTURE repository that the EXAMCARDS_WIN and GYROVIEW_WIN components commonly read and write, which may determine a data dependency between these components. The matrix (c) is used to find the parameter (data element in the Configuration Repository), that the SCANNER component constantly reads during the tasks in the scenario, which may

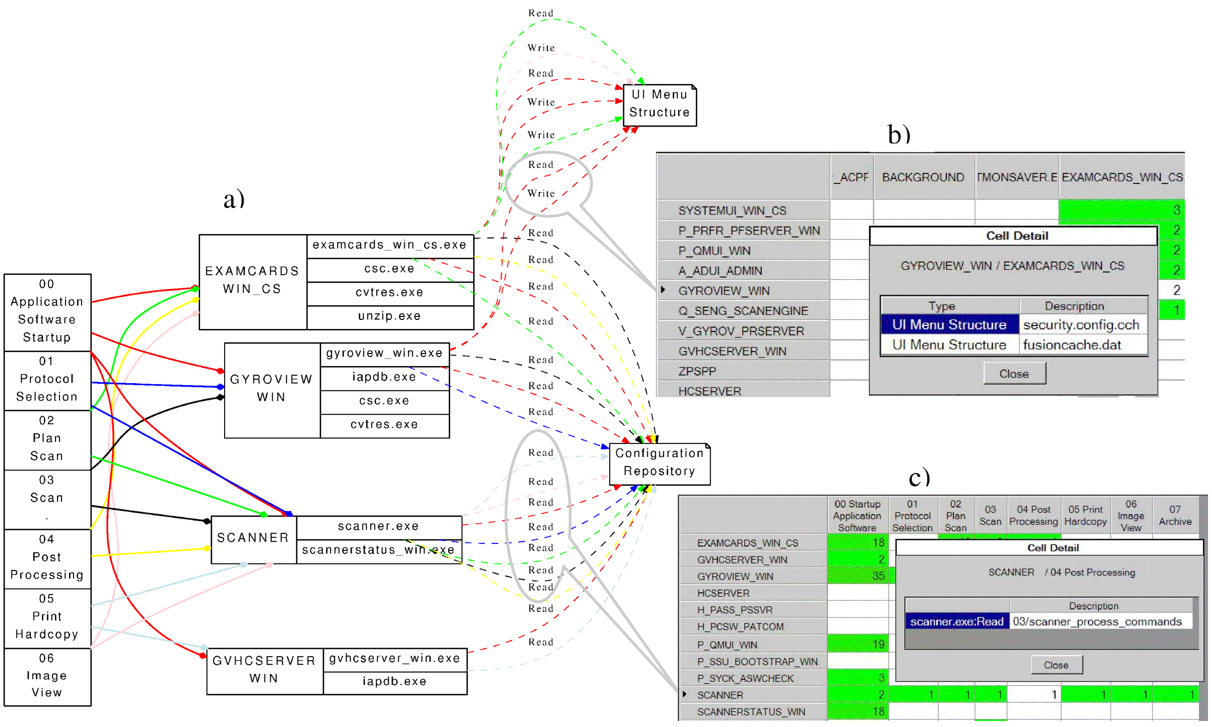
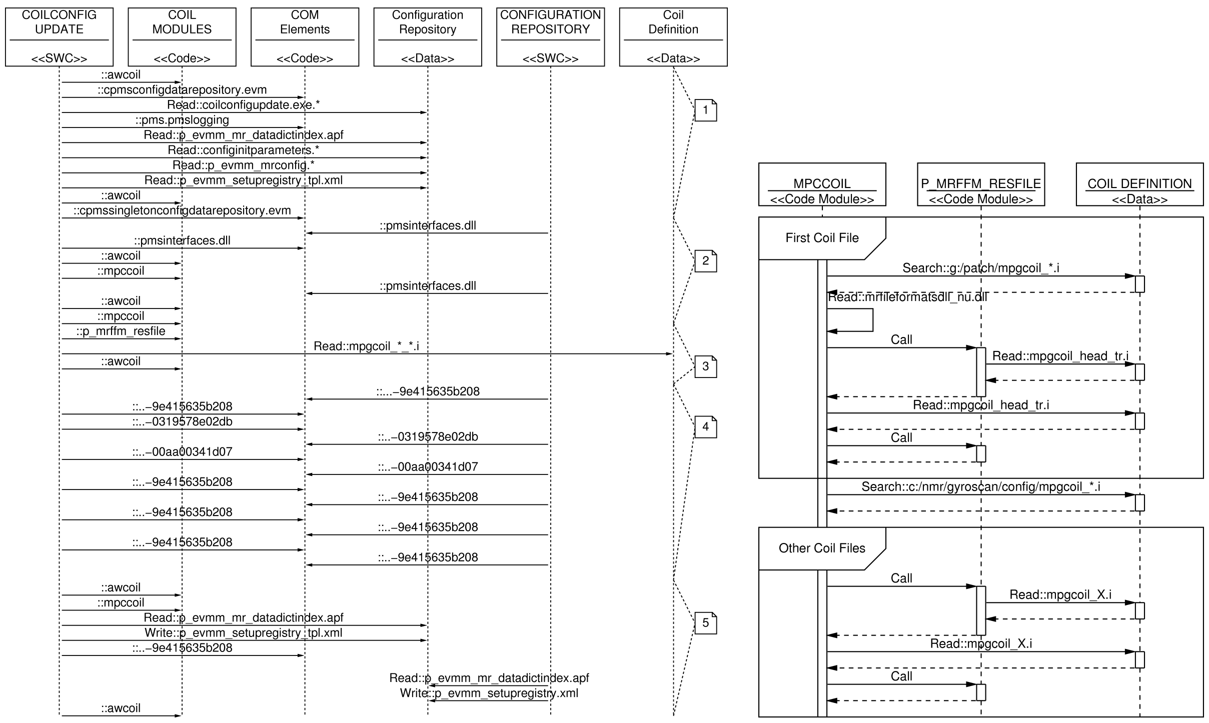


Figure 3.13: *Looking for details with functional mapping and matrix models to determine dependencies*

determine a dependency between the component and the data repository.

Sequence diagrams constitute another kind of model that provides details that complement high-level information and can be used to determine dependencies, especially due to temporal relationships. Figure 3.14 illustrates two sequence diagrams that we constructed to zoom in on details, e.g. the code elements that software components execute before accessing, reading or writing, to data repositories. The sequence diagram in (a) aims at zooming in on the sequence of runtime activities that realize the task Add Coil of the scenario described in Figure 3.12. The elements described in the sequence diagram are the key software components (stereotyped as SWC) that execute or load code elements (stereotyped as Code) such as modules, COM elements, and DLLs, before accessing data repositories (stereotyped as Data). The names in the edges between the aggregations in the sequence diagram contain the respective runtime activity and the element of the aggregation that is used.

When the problem at hand requires zooming in on further details and the collected runtime data contains such details, it is possible to construct sequence diagrams with finer runtime information. The notes in the right side of the sequence diagram (a) describe that the task can be dived into five subtasks: operator actions (1), COM communication (2, 4), load Coils' configuration (3), and update system repository configuration (5). The sequence diagram (b) in Figure 3.14 zooms in on the third subtask (3). The diagram shows the runtime activity and the order required to read all the files that contain the Coil Definition data, a key aspect for the feature under analysis. The sequence diagram shows, at the top, the realization to read a first file and then how a similar sequence is repeated to read subsequent files.


COILCONFIG UPDATE <<SWC>>
COIL MODULES <<Code>>
COM Elements <<Code>>
Configuration Repository <<Data>>
CONFIGURATION REPOSITORY <<SWC>>
Coil Definition <<Data>>
::awcoil
::cpmsconfigdatarepository.evm
Read::coilconfigupdate.exe.*
::pms.pmslogging
Read::p_evmm_mr_datadictindex.apf
Read::configinitparameters.*
Read::p_evmm_mrconfig.*
Read::p_evmm_setupregistry_tpl.xml
::cpmssingletonconfigdatarepository.evm
::pmsinterfaces.dll
::mpccoil
::p_mrffm_resfile
Read::mpgcoil_*_*.i
::...−9e415635b208
::...−0319578e02db
::...−00aa00341d07
Write::p_evmm_setupregistry_tpl.xml
1
2
3
4
5
MPCCOIL <<Code Module>>
P_MRFFM_RESFILE <<Code Module>>
COIL DEFINITION <<Data>>
First Coil File
Search::g:/patch/mpgcoil_*.i
Read::mrifileformatsdll_nu.dll
Call
Read::mpgcoil_head_tr.i
Search::c:/nmr/gyroscan/config/mpgcoil_*.i
Other Coil Files
Read::mpgcoil_X.i

(a) (b)

Figure 3.14: *Looking for details with sequence diagrams models*

3.6.3 Conformance and realization analysis.

Large and complex software-intensive systems like the Philips MRI have strict constraints on nonfunctional properties such as reliability and performance. Ideally, the architecture and design, documented or in the mental models of the experts, specify how to achieve the desired quality. However, the realization deviates from the specification, especially when the implementation includes third party or off-the-shelf components, multiple implementation technologies, and it is constrained by the characteristics of the runtime platform. An execution profile can convey a number of insights about the runtime and realization of a system like the Philips MRI scanner, which practitioners can use to refresh their mental models and validate their expectations about the realization of the system. Execution workflow models and some types of matrix models are the main means provided by an execution profile for conformance and realization analysis.

An execution workflow model is a Gantt-chart-based representation that describes how the tasks that realize a given feature are distributed over time and over the processing nodes of the system. Figure 3.15 shows a example of an execution workflow model and its respective notations. The model describes the major tasks that are performed inside the main computers of the Philips MRI scanner (Host, Reconstructor, DAS) during the startup of its software system. The description helped the organization to visualize the actual realization of this key feature, i.e., the actual duration of each task and how these tasks run concurrently or sequentially across the processing nodes of the system. In addition, the description shows how major tasks can be mapped to finer tasks to zooming into details. For example, PF Client Recon, which is a major task in the Host, coordinates the start-up of the Reconstructor subsystem and is mapped to the finer tasks in the Reconstructor computer. Currently, the model is constructed on a regular basis to verify and predict changes in the performance, e.g. distribution and duration of the startup process through ongoing and future development projects.

Figure 3.16 shows two matrix models that correspond to the type IV listed in Table 3.3. The matrices complement the functional mapping model in Figure 3.12, by checking the expectations about the role of key components in the scenario. On the one hand, matrix (a) serves to check the actual processes, per software component, that are active in the scenario. On the other hand, matrix (b) helps to check the actual number of threads, per software component, during the respective tasks in the scenario. The latter enabled the identification of unintended use of threading mechanisms. As the figure shows, the situation was especially noticeable in the realization of the Enable Coil task. After further investigation as part of a downstream analysis, the situation was attributed to the implementation of third-party elements in the SMARTCARD_WIN component.

In addition to matrix models, functional mapping models help to distinguish details such as the implementation technology of software components and the use of platform utilities. For example, the model in Figure 3.12 shows that two of the software components in the scenario include different instances of the csc.exe process, which represents the C-Sharp compiler for the systems runtime platform. This shows that the respective software components include or use .Net implementations. The same model shows that a software component contains an attrib.exe process, and the model in Figure 3.13 shows that another component contains an unzip.exe processes, which indicates that these components require platform utilities to manage access rights and data compression respectively.

Table 3.3: *Type of matrix models for runtime information analysis.*

Type	Rows	Columns	Cells
I	Soft. components	Soft. components	Data, code, and platform elements
II	Tasks	Tasks	Data, code, and platform elements
III	Soft. components	Tasks	Components' interactions
IV	Soft. components	Tasks	Components' processes or threads

3.7 Contribution and potential limitations

In this section, we describe the technical contribution of our and some possible limitations.

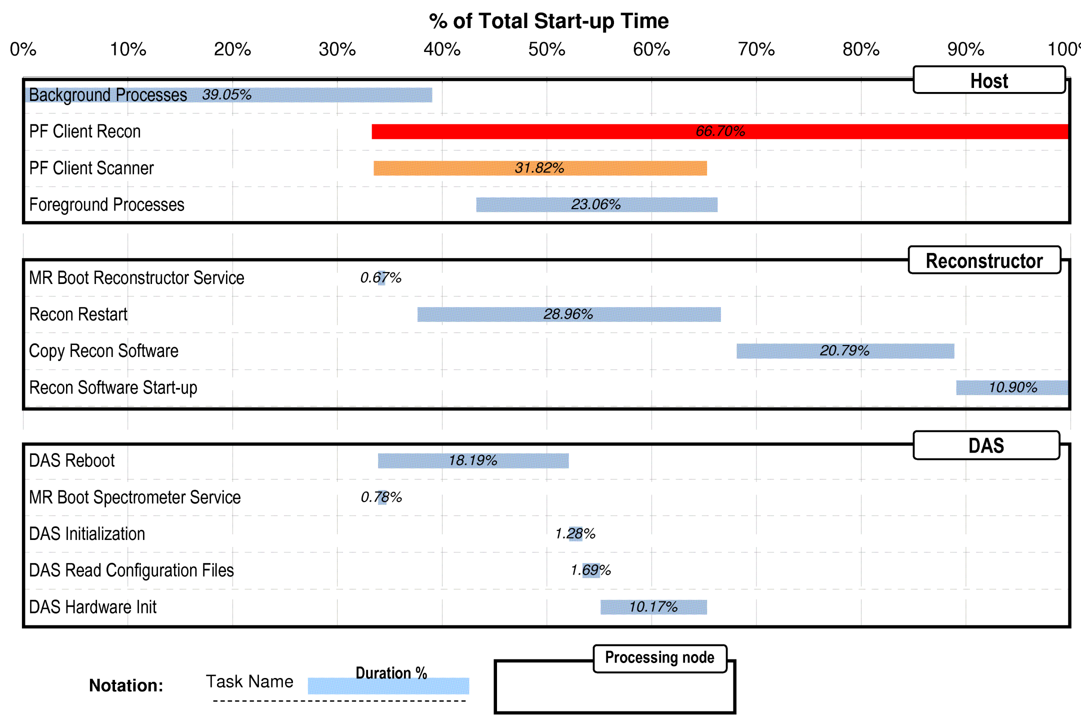
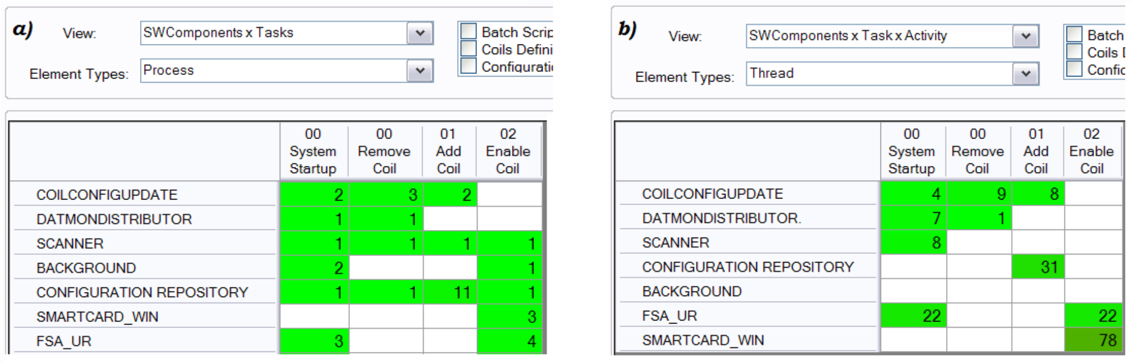


Figure 3.15: *Execution workflow of the start-up process of the Philips MRI scanner*


a) View: SWComponents x Tasks
Element Types: Process

☐ Batch Scrip
☐ Coils Defini
☐ Configurati

	00 System Startup	00 Remove Coil	01 Add Coil	02 Enable Coil
COILCONFIGUPDATE	2	3	2	
DATMONDISTRIBUTOR	1	1		
SCANNER	1	1	1	1
BACKGROUND	2			1
CONFIGURATION REPOSITORY	1	1	11	1
SMARTCARD_WIN				3
FSA_UR	3			4

b) View: SWComponents x Task x Activity
Element Types: Thread

☐ Batch
☐ Coils I
☐ Confic

	00 System Startup	00 Remove Coil	01 Add Coil	02 Enable Coil
COILCONFIGUPDATE	4	9	8	
DATMONDISTRIBUTOR.	7	1		
SCANNER	8			
CONFIGURATION REPOSITORY			31	
BACKGROUND				
FSA_UR	22			22
SMARTCARD_WIN				78

Figure 3.16: *Examples of matrix models to check the realization of the Philips MRI scanner*

3.7.1 Technical contribution

As part of our research, we have observed that architecture reconstruction is an intrinsic activity for the evolution of software-intensive systems. Architecture reconstruction solutions have become the basis for re-documentation, investigation, and system understanding, especially for practitioners who need a global understanding of the system. Architecture reconstruction can make use of resource such as available documentation, interviews, domain knowledge, source code, and as we demonstrated, logging and runtime measurements. A number of solutions, e.g. approaches and techniques, have been proposed in the literature to support architecture reconstruction (Koschke 2009, Ducasse and Pollet 2009). Yet, nowadays software-intensive-systems like the Philips MRI scanner pose a number of challenges that architecture reconstruction solutions must cope with to effectible contribute to the maintenance and evolution of such systems.

Stoermer et al. surveyed architecture reconstruction practice needs and approaches (Stoermer et al. 2002). Their findings with respect to practice needs are very similar to some of the aspects that we addressed through the development and validation of our approach. To distinguish the technical contribution of our approach, we present the practice needs, as described by Stoermer et al., and how we address them for the Philips MRI scanner.

- **Multiple Views**. One of the first problems to perform architecture reconstruction activities is to determine which architecture views sufficiently describe the system and cover stakeholder needs. The evaluation by Stoermer et al. describes that none of the approached in their evaluation supports an explicit selection of architecture views that can be systematically reconstructed in order to describe a system sufficiently and address its stakeholders' needs. *The support of our approach* for multiple views relies on using execution viewpoints. As we described in Section 3.3.1, execution viewpoints explicitly describe the set of views that can be constructed with our approach, as well as the concerns that can be addressed by constructing a given execution view. The validation of our approach, described in Section 3.5, shows that a view is selected based on the characteristic of the problem at hand, i.e., the concerns about the realizations of a given set of features to be changed. Then the view is systematically constructed in a top-down fashion, i.e., with high-level information at first and details when needed by the stakeholders. As a whole, our approach is a solution that offers an explicit catalog of views and a systematic process to select and construct a required view.

- **Enforce Architecture**. On of the main reasons to conduct architecture reconstruction activities is the lack of consistency between the as-built architecture and the as-designed architecture of a system, because traceability information is missing, from architecture design through code implementation. *The support of our approach* to enforce architecture relies on a metamodel described in Section 3.3.2. The metamodel makes explicit the concepts or entities and traceability relationships between them. The development organization and we agreed on that the metamodel is the bases to describe and analyze the runtime architecture of the Philips MRI scanner. Therefore, during the validation of our approach, with the metamodel as a common guideline, some of the models were dedicated to check how the realization of key features match the expectations or mental models of the practitioners.

- **Quality Attribute Changes**. Another reason to conduct architecture reconstruction on software-intensive systems is to determine the relationship among quality attributes and architecture elements. Software-intensive systems like the Philips MRI scanner are tightly coupled with performance and distribution requirements. These characteristics can change more often than other system aspects due to dependencies, not only with software elements but also with the hardware elements. *The support of our approach* includes the construction of models to determine the relations between performance and both, software and hardware elements. The validation of our approach, presented in Section 3.5, shows that some of the execution models constructed with our approach are dedicated to support the description, analysis, and assure the performance of a key feature like the start-up of the system. In addition, the validation of our approach presented in (Callo Arias, America and Avgeriou 2009a) shows that our approach supports the construction of execution models to describe and analyze the performance of data-intensive and computation-intensive features of the Philips MRI scanner.

- **Common and Variable Artifacts**. Commonality and variability are used in product line environments so that organizations can reduce costs by reusing common assets. The problem is to identify the common and variable parts in several similar products. Variability is important for the Philips MRI scanner. However, we have not observed direct application or results regarding variability as part of the development and validation of our approach.

- **Binary Components**. The software industry is quickly moving toward systems based on commercial components. A component in this context has three characteristics: it is produced by a vendor, who sells the component or licenses its use; it is released by a vendor in binary form; and it offers an interface for third-party integration. The problem is conducting architecture reconstruction in settings where commercial off-the-shelf (COTS) components are used. *The support of our approach* to conduct architecture reconstruction when COTS components are used relies on the kind of abstractions and the source of information that are used. The Philips MRI scanner has a large implementation, which contain both in-house and COTS components. Though the source code is available for most COTS components, the time and knowledge required to understand them are simple not available during development projects. Therefore, practitioners like architects and designers look at these components as gray and even black boxes. In the validation of our approach, presented in Section 3.5, COTS components are mapped to entities, like runtime processes or code elements, e.g., DLLs, COM components, and code modules, which are executed inside threads.

- **Mixed-Language**. Software systems implemented in several languages are commonplace today. The problem is to reconstruct the architecture of a system that is implemented in more than one language. The Philips MRI scanner has a heterogeneous implementation. Thus, it was required that our approach can abstract away or cross the borders between elements that are implemented with different programming languages and paradigms. *The support of our approach* to deal with heterogeneous implementations relies in the construction of high-level descriptions at first, and then, if needed, details for dedicated areas. In the validation of our approach, presented in Section 3.5, we show that high-level descriptions present software components as a set of processes, which abstract away their implementation language and paradigm. However, as we described for realization analysis, in some cases the description of software components provides means to distinguish the implementation technology.

- **Overhead**. In addition to the practice needs reported in (Stoermer et al. 2002), 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. In the case of our approach, the overhead was minimal. We use already available data from logging and runtime measurements. As we described in Section 3.4.1, logging is a source of runtime information created and maintained by development organization. Although it is based on a sort of source code instrumentation, the validation of our approach did not required extra instrumentation for logging. Instead, we choose to complement the existing logging, with runtime measurements collected with monitoring tools provided by the system runtime platform.

3.7.2 Potential limitations

The usage of viewpoints and a 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. 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 (Callo Arias et al. 2008, Callo Arias, America and Avgeriou 2009b, Callo Arias, America and Avgeriou 2009a) illustrate how other practitioners and researchers can perform such customization for their particular settings. This 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 (Callo Arias, America and Avgeriou 2009b). Then the execution metamodel may include similar high-level elements, 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 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 3.4 and Section 3.5, 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, i.e., 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 Philips MRI scanner 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 practitioners 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.

3.8 Related Work

In this section, we discuss work that is related to the main characteristics of our approach.

3.8.1 Top-down solutions

Top-down solutions take some high-level knowledge as input, e.g. problem requirements, decomposition of a system, and design conventions. This input guides a discovery of the architecture or the information needed to address the problem at hand. Software reflexion models (Murphy et al. 2001) is the most representative example of a top-down solution closely related to our work. The top-down process for reflexion models summarizes a source model, i.e. the source code of a software system from the perspective of a particular high-level model, which is a hypothesis defined by a developer. The summarization is based on declarative mapping that associates entities in the source code with entities in the high-level model defined by the developer.

The use of high-level knowledge and declarative mapping are the main commonalities between reflexion models and our approach. In the case of our approach, the use of high-level knowledge is driven by a set of reusable viewpoints and a metamodel about the runtime of a system. Similarly, our mapping rules associate runtime data, i.e. logging and measurements, and high-level abstractions with respect to the runtime structure and behavior of the system. We consider that a combination of the process to construct reflexion models and our approach can be an ideal top-down solution to construct module and execution views, especially for organizations that need to manage size and complexity of software-intensive system.

3.8.2 Runtime description and analysis

The main goal of our approach is to support the description and analysis of the runtime of a system like the Philips MRI scanner. In the literature, a number of techniques focus on the description and analysis of the runtime of software systems. According to the kind of views that can be constructed, the techniques can be classifies as follows:

- *Dynamic module views:* Dynamic module views describe the runtime interactions between implementation artifacts such as modules, classes, and objects. These views are mainly constructed applying dynamic analysis techniques (Briand et al. 2003, Hamou-Lhadj and Lethbridge 2004, Systä 2000, Egyed 2003). These techniques focus on the extraction and abstraction of execution traces, which track code level events at runtime, e.g., the entry and exit of functions and methods. To extract execution traces, these techniques rely on techniques such as compiler profiling, or source code instrumentation (Briand et al. 2003). For the abstraction, i.e., presenting the extracted data at a high-level of abstraction and manage large amounts of execution traces, these techniques use aggregation, summarization, and visualization mechanisms (Hamou-Lhadj et al. 2005, Safyallah and Sartipi 2006, Cornelissen et al. 2007).

  Code-based techniques are especially useful when the problem at hand involve concerns around the implementation structure or require accurate traceability to code elements, e.g. modules or at least functions. The development of dynamic analysis techniques is overemphasized for object-oriented implementations (Cornelissen et al. 2009). In the case of our approach, the proposed dynamic analysis technique is more suitable for distributed and multithreaded systems with heterogeneous implementations like the Philips MRI scanner, where analyzing only the source code is not enough to achieve a proper understating of the system.

- *Application management views:* Management views describe information related to the externally observable behavior of system elements (Keller and Kar 2000) such as performance, availability, and other end-user-visible metrics (Brown et al. 2001). Management views of software systems can be constructed using techniques that analyze monitored information and system repositories (Keller and Kar 2000, Brown et al. 2001, Gupta et al. 2003). 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.

  Management views describe the major elements of a running system (subsystems, applications, services, data repositories etc.) as black boxes. This enables a high-level understanding of the runtime of a system, the integration of constructed views into the system documentation, and their reuse in further analysis (Brown et al. 2001). The focus on high-level abstractions and the use of monitored information are important aspects that inspired the development or our approach. Therefore, our approach can be seen as an extension of application management techniques, especially for situations where unintended variations of end-user-visible properties need to be analyzed and solved by software architects and designers as part of the development and maintenance cycle.

3.9 Conclusions and Future Work

The contribution of our approach is centered on two points. First, it is a structured and problem-driven architecture reconstruction 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 and scalable solution to construct execution views. In our future work, we aim to report further validation of our approach with 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.