The Journal of Systems and Software 84 (2011) 1447–1461

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Defining and documenting execution viewpoints for a large and complex software-intensive system

Trosky B. Callo Arias [a,*], Pierre America [b], Paris Avgeriou [a]

[a] Department of Mathematics and Computing Science, University of Groningen, Nijenborgh 9, 9747 AG Groningen, The Netherlands
[b] Philips Research and Embedded Systems Institute, The Netherlands

ARTICLE INFO

Article history:
Received 11 June 2010
Accepted 18 November 2010
Available online 25 November 2010

Keywords:
Viewpoints
Views
Architecture
Runtime

ABSTRACT

An execution view is an important asset for developing large and complex systems. An execution view helps practitioners to describe, analyze, and communicate what a software system does at runtime and how it does it. In this paper, we present an approach to define and document viewpoints that guide the construction and use of execution views for an existing large and complex software-intensive system. This approach includes the elicitation of the organization's requirements for execution views, the initial definition and validation of a set of execution viewpoints, and the documentation of the execution viewpoints. The validation and application of the approach have helped us to produce mature viewpoints that are being used to support the construction and use of execution views of the Philips Healthcare MRI scanner, a representative large software-intensive system in the healthcare domain.

1. Introduction

The usage of several architectural views is a common practice to construct and document the architecture of large software-intensive systems (Hofmeister et al., 2007; ISO/IEC-JTC1/SC7, ISO/IEC 42010, 2007). The IEEE 1471 standard and its successor, the ISO/IEC 42010 standard provide a widely accepted conceptual definition of architectural views, viewpoints and models (ISO/IEC-JTC1/SC7, ISO/IEC 42010, 2007):

- *An architectural view* is a representation of a set of system elements and relations associated with them, conforming to a specific viewpoint.
- *An architectural viewpoint* frames particular concerns of the system stakeholders and consists of the conventions for the construction, interpretation, and use of an architectural view.
- A view may consist of one or more *architectural models*. Each such architectural model is developed using the conventions and methods established by its associated viewpoint. An architectural model may participate in more than one view.

As part of our research on the evolvability of large software-intensive systems (van de Laar et al., 2007), we observed that suitable architectural views are indispensable assets to improve and sustain the evolvability of systems (Muller, 2004, 2009). Such views help practitioners to understand the existing system, to plan and evaluate intended changes, and to communicate them to others efficiently. In particular, we are interested in *execution views*, which consist of a set of models that describe and document *what a software system does at runtime and how it does it*. The term runtime refers to the actual time that the software system is functioning during testing or in the field.

The runtime behavior and structure of a software-intensive system as well as their evolution can be particularly complex. The software part of a software-intensive system can have heterogeneous implementations and consist of multiple processes, each with multiple threads, and deployed across several computers. Consequently the runtime of software-intensive systems can change more often than other system aspects due to its complex dependencies not only with the software elements but also with the hardware elements. In addition, when the runtime of a system is tightly constrained by performance and distribution requirements, tuning any of the system characteristics to align with changing requirements will likely change the runtime of the system as well. Therefore, up-to-date execution views are an essential prerequisite to understand the complexity of software-intensive systems and be prepared to respond effectively to change.

However, to the best of our knowledge describing the runtime of software-intensive systems has not received enough attention. This prompted us to focus particularly on supporting practitioners on how to construct execution views for large and complex software-intensive systems. There are two ways to construct architectural views: either reuse the guidelines of predefined viewpoints avail-

* Corresponding author.
E-mail addresses: trosky@cs.rug.nl (T.B. Callo Arias), pierre.america@philips.com (P. America), paris@cs.rug.nl (P. Avgeriou).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.11.908

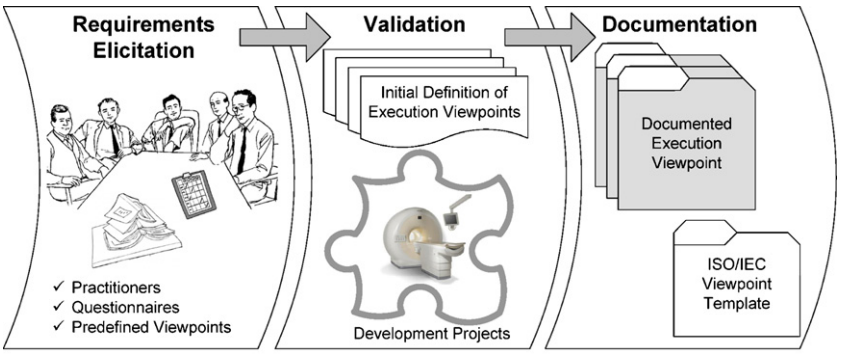


Fig. 1. Defining and documenting execution viewpoints.

able in the literature (e.g. Clements et al., 2002; Hofmeister et al., 1999; Muller, 2004; Rozanski and Woods, 2005) or define new ones. Often, for software intensive-systems, the predefined viewpoints are not a good match, due to the numerous specific concerns of the particular stakeholders. Therefore, one has to define customized viewpoints to frame the concerns of the stakeholders at hand.

In this paper, we present an approach to define execution viewpoints for organizations developing large and complex software-intensive systems through three phases (see Fig. 1). The first phase includes the identification of predefined viewpoints in the literature and the elicitation of the organization's requirements for execution views. The organization's requirements are derived by observing and interviewing key practitioners with dedicated questionnaires. The second phase includes the initial definition and validation of a set of execution viewpoints. The initial definition of the execution viewpoints takes place by considering the concerns and requirements that motivate the creation of new viewpoints or the customization of predefined viewpoints. The validation focuses on the application and tuning of the initial definition across development projects. The third phase is the documentation of execution viewpoints, which contain reusable knowledge that guided the construction and use of execution views during validation.

We have applied this approach as part of the documentation of the execution architecture of a Magnetic Resonance Imaging (MRI) scanner. This system is a representative large and complex software-intensive system, developed by Philips Healthcare MRI (Philips Healthcare: Magnetic Resonance Imaging, 2010). The defined execution viewpoints have been used and validated in a number of development projects. The viewpoints are currently considered mature and are being used to support the construction and use of execution views for different parts of this system. We expect that other organizations and researchers can reuse our approach to construct other execution viewpoints, or other types of viewpoints. In addition, practitioners can reuse the execution viewpoints that we define and document in this paper.

The organization of the rest of this paper follows the steps of the proposed approach. In Section 2, we summarize how we identified a few predefined viewpoints from the literature. In Section 3, we describe how to elicit the requirements of a particular development organization interviewing key practitioners. Section 4 summarizes the initial definition of execution viewpoints and the validation. In Section 5, we describe and present the documentation of the execution viewpoints. Finally, in Section 6, we provide some conclusions.

The elaboration of the validation and the documentation phases of the approach are the main extensions to the previous presentation of the approach in Callo Arias et al. (2009). Other changes include the introduction of the approach and the summary of the initial definition of execution viewpoints.

2. Predefined execution viewpoints

In this section, we describe our motivation to search for predefined viewpoints and the result of our search.

2.1. Motivation

To define specific execution viewpoints, we searched the literature for predefined viewpoints that address somehow what a system does at runtime and how it does. In doing so we conform with the conceptual model from the ISO/IEC 42010 standard (ISO/IEC-JTC1/SC7, ISO/IEC 42010, 2007). Fig. 2 illustrates the part of the conceptual model that describes the definition of specific viewpoints, the concepts of viewpoints, views, and models with respect to execution. According to this model an execution viewpoint can cite a predefined viewpoint, in the sense that the former can be defined reusing (customizing or extending) the latter.

2.2. Identified predefined viewpoints

Our literature search for predefined viewpoints resulted in the identification of five candidates, which are summarized in Table 1. To the best of our knowledge, out of all possible candidate viewpoints, the selected set has the most comprehensive and elaborated description for use as predefined execution viewpoints. The summary describes the names of the viewpoints as presented in the literature, the set of concerns framed by the viewpoints, and the kind of system elements used in the models of these viewpoints.

These predefined viewpoints can be classified into two groups based on their concerns. The first group includes:

- The *concurrency viewpoint* of Rozanski and Woods (2005), which describes the concurrency structure of the system, mapping functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently.
- The *behavior description* of Clements et al. (2002), which proposes a language-independent way to document behavioral aspects of the interactions among system elements.

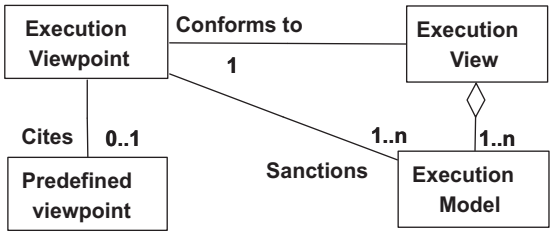


Fig. 2. Reuse of predefined viewpoints for an execution viewpoint.

Table 1
Predefined viewpoints for execution views.

Viewpoint	What it describes (concern)	System elements
Concurrency (Rozanski and Woods, 2005)	Task structure and mapping of functional elements to tasks Inter-process communication and state management Synchronization and integrity Start-up, shutdown, task failure, and reentrancy	Processes, process groups, threads, inter-process communication
Behavior description (Clements et al., 2002)	Types of communication Constraints on ordering Clock-triggered stimulation	Use cases, structural elements, processes, states, applications, and objects
Deployment (Rozanski and Woods, 2005)	Hardware required (specification and quantity) Third-party software requirements and technology compatibility Network requirements and capacity and physical constrains	Processing and client nodes, network links, hardware components, and processes
Deployment style (Clements et al., 2002)	Allocation, migration, and copy relations between software elements and computing hardware Properties of computing hardware, e.g., bandwidth, and resource consumption	Software elements (processes) and computing hardware (processor, memory, disk, etc.)
Execution architecture (Hofmeister et al., 1999)	Execution configuration and its mapping to hardware devices Dynamic behavior of configuration Communication protocol Description of runtime entities and their instances	Processes, tasks, threads, clients, servers, buffers, message queues, and classes

The second group includes:

- The *deployment viewpoint* of Rozanski and Woods (2005), which addresses how to describe the environment into which the system will be deployed including the dependencies the system has with its runtime environment.
- The *deployment style* of Clements et al. (2002), which also addresses how to describe the allocation of components and connectors to execution platforms.

In addition, another predefined viewpoint is the execution architecture of Hofmeister et al. (1999), which spans the two groups, describing the mapping of functionality to physical resources and the runtime characteristics of the system.

3. Eliciting the organization's requirements for execution views

Eliciting the concerns of stakeholders is of paramount importance, in order to choose appropriate views (Clements et al., 2002) that frame these concerns and identify which views to recover from an existing system (van Deursen et al., 2004). In order to identify the requirements for execution views, we conducted a series of interviews with key experts of our industrial partner using specific questionnaires. In this section, we summarize the key aspects of the questionnaire design and interviews, and the elicited concepts and concerns.

3.1. Questionnaire design

The main goal of the specific questionnaires was to collect information on which execution views to create, what to describe in a particular model, how to choose the abstraction level, and how it should be described. Often, asking these broad questions to practitioners does not provide precise or useful answers. To overcome this, we designed two types of questionnaires (overview and model-specific). To design them, we summarized predefined viewpoints in the literature and our own research observations, and applied guidelines on reviewing software architecture descriptions (Obbink et al., 2008).

Overview questionnaires help us to estimate the value of an execution viewpoint and get an insight on how a given interviewee may use it. To focus the questionnaire, we centered the questions on a set of existing documents containing some execution models that the interviewee created or use often.

Model-specific questionnaires help us to assess how a specific execution model created or often used by the interviewee aligned to descriptions of similar models of predefined viewpoints. Thus, with each model-specific questionnaire we attached at least two models: the one used or created by the interviewee and a related example from the literature. Table 2 summarizes the group of questions for both types of questionnaires, overview and model-specific. For an example of a full questionnaire, see Appendix I.

3.2. Interviews

To conduct the series of interviews, and keep them manageable and productive, it is necessary to identify a set of representative practitioners. We initially involved two stakeholders of the development organization who are actual consumers and producers of execution views. First, a senior designer who documented an execution view in the past using as a main reference the 4 + 1 View Model (Kruchten, 1995) aiming to support the analysis of the system performance. Second, an architect in charge of architecting and designing software interfaces for system-specific hardware devices. Later, we selected additional stakeholders who were mentioned as major contributors or actual users of the chosen document

Table 2
Questionnaires structure.

Group of questions	Overview	Model-specific
1. Authors and contributors	X	X
2. Creation and maintenance	X	X
3. Intended and actual users	X	X
4. Usage in daily activities (predefined viewpoint)	X	X
5. Usage in other activities (observations and experience)		X
6. Description of concerns (predefined viewpoint)		X
7. Representation language and level of detail		X

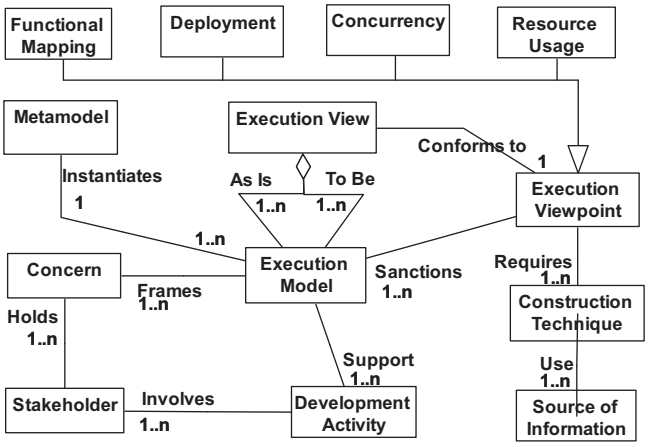


Fig. 3. Conceptual model to define execution views and viewpoint.

for the interview, e.g., other software architects, designers, platform support engineers, and managers. After conducting an interview, we validated the collected information sending the questionnaire (with answers and comments) to the interviewee who corrected and sometimes extended the captured information.

3.3. Identified concepts and concerns

Through the series of interviews, we identified a set of concepts and relationships between them. Fig. 3 illustrates the concepts and their relationships. This conceptual model is based on the model presented by the ISO 42010 standard (ISO/IEC-JTC1/SC7, ISO/IEC 42010, 2007), but here we limit ourselves to execution views, models, and viewpoints instead of general architectural views, models, and viewpoints from the standard. The functional mapping, deployment, concurrency, and resource usage viewpoints are specific viewpoints that we define and document in Sections 4.1 and 5 respectively. In addition, we include concepts such as development activity, metamodel, and construction technique to illustrate how execution views and viewpoints fit within the development process and the organization. In the rest of this section, we focus on the descriptions of the main concepts (execution model and metamodel) and the identified major concerns related to system evolvability within development activities. Construction techniques and sources of information are presented in our previous work (Callo Arias et al., 2008).

3.3.1. Execution models

From the results (answers and comments) of questions in groups 1–4, we identified that a development organization often needs to construct '*As Is*' and '*To Be*' execution models to build an execution view. The concept of '*As Is*' and *To Be*' are also applicable to models of other architectural views, but to keep the focus of this paper, we describe these concepts for models of an execution view.

'*As Is*' models describe the execution of the current system. These models are often created to support the acquisition of knowledge about key execution scenarios or the interactions between key system components. A '*To Be*' model describes the execution of a system that does not yet exist. Such models are typically created to design and evaluate one or more alternatives for a future system and to communicate the chosen alternative to the implementers. After implementation, a new '*As Is*' model can be created and compared to the chosen '*To Be*' model. Since nowadays a system is rarely ever designed from scratch but is typically based on existing systems (i.e. Brownfield site, Hopkins and Jenkins, 2008), it is often a good idea to construct a '*To Be*' model by modifying or taking as a reference an existing '*As Is*' model.

3.3.2. Metamodel of system execution elements

When identifying the information needs of the practitioners, we found it very useful to describe the various elements that play a role in system execution in a metamodel, which defines a number of concepts that occur in the execution models. Fig. 4 shows such a metamodel with system execution elements and relationships between them. We developed this in our earlier work (Callo Arias et al., 2008) and validated and refined it during the interviews. Most predefined viewpoints (see Table 1) also use several of these elements, e.g., processes and threads, to create execution models. Our metamodel extends the concepts of the predefined viewpoints, including elements and relationships to address the organization's requirements that we identified to construct execution views of a large software system. The particular extensions that we introduce are elements such as execution scenario, task, software component, and activity. These extensions are meant to cope with three major issues: complexity and size of the system, explicit links with other system views, and analysis of resource usage. In Section 5, we describe these extensions in more detail in the discussion of the identified viewpoints. We also provide a detailed description of the elements and relationships of this conceptual model in Callo Arias et al. (2008).

Note that the metamodel does not apply to an individual execution model, but is *shared* among the execution models. In this way, it indicates important relationships between the models and can help to establish consistency among the models. We expect that using a single, shared metamodel not only in the execution views but also across all architectural views may contribute significantly to their mutual consistency.

3.3.3. Organization concerns related to system evolvability

Based on the result of questions in groups 2–5, we found that the construction of execution models is a goal-driven and often problem-driven activity to evolve an existing system. This means that the concerns of the stakeholders relate to the activities they perform within a given development project towards specific goals. Table 3 lists the major organization wide concerns and the development activities that need support of execution views, which we elaborate in the following paragraphs:

- *System understanding*: In addition to the result of questions in groups 2–5, our own observations helped us to identify two

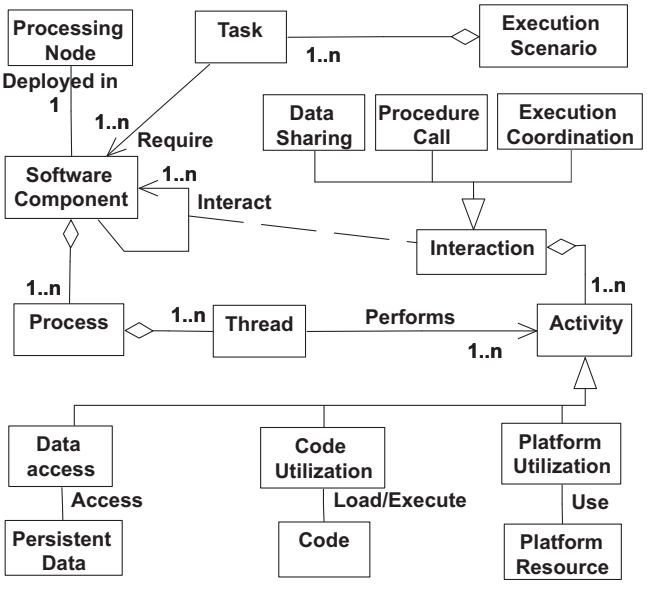


Fig. 4. Metamodel of system execution elements.

Table 3
Organization concerns and development activities supported by execution models.

Concern	Development activity
System understanding	Education and training, dependency analysis, and corrective maintenance
Project planning	Analysis of alternative and future architectures and/or designs
Communication	Between development units or teams and with customers and providers
Conformance of design and implementation	Architecture documentation, verification of non-functional requirements, and testing

aspects of how an execution view supports acquisition of system knowledge. On the one hand, execution models support system-specific education and training of new developers. Often new developers are exposed to execution models before they can start reading and writing code. This practice helps new developers to create a mental model of the overall system, the system components they develop, and their relations (dependencies) with the rest of the system components. On the other hand, '*As is*' execution models help *all* practitioners to constantly refresh, validate, and extend their mental models, in particular to support system corrective maintenance activities that aim to improve the existing run-time structure and manage unpredicted system behavior.

- *Project planning*: Practitioners need to construct '*To be*' execution models to support two particular activities. On the one hand, these models are needed to distinguish and analyze the difference between considered alternative or future architectures and designs that aim to improve quality attributes such as reliability (Sozer and Tekinerdogan, 2008), dependability, and safety (Hunt et al., 2007). This is important, as it is often not obvious how the realization of the alternative design may affect the structure and behavior of the system at runtime, and therefore influence other system quality attributes. On the other hand, as we described in Section 4.1, execution models are necessary to describe the overall system structure, its components, and their interactions that make up the system functionality of interest. Often system components are mapped to development units within or outside the organization. Thus describing the involved system components enables the identification of the involved units, and therefore the planning and budgeting of responsibilities, if possible, as a downstream process.
- *Communication*: Another goal of describing the architecture of a software system is to support the communication between system stakeholders. In particular, we identified that besides the mental models that practitioners may have, they need explicit evidence in a common language (i.e. diagrammatic representations of execution models) to supports three links of communication within the development organization. First, execution models are useful to transfer technical knowledge of the system design and implementation. This supports the communication of designers and developers with architects and managers. Second, execution models are needed to describe how the system uses third-party components at runtime. These models will enable the communication of development units (external or internal) with customer designers, developers, and testers. Third, execution models are needed to describe how the software system interacts with and uses the resources of its runtime platform. These models will enhance the communication of the design and implementation units with the (internal or external) unit supporting the system runtime platform.
- *Conformance of design and implementation*: Large and complex software-intensive systems have strict constraints on their non-functional properties such as reliability, safety, and performance. Ideally, the architecture and design should describe how to achieve those requirements, but often the implementation deviates from these requirements at runtime. This usually happens when the implementation uses third party or off-the-shelf components, facilities provided by the implementation technology and the runtime platform, such as dynamic loading of shared libraries, plug-in mechanisms, and mechanisms to manage memory access. Thus, to verify non-functional requirements and properly test the system, it is often necessary to construct '*As is*' execution models to describe changes in the access and utilization of resources such as shared memory, shared code libraries, communication paths, and power consumption. Thus, '*To be*' models can be updated, extended, and analyzed.

4. Initial definition and validation of execution viewpoints

The main findings (requirements and observations) from questions in groups 5–7, and the identified concepts and concerns (see Section 3.3) have lead us to the creation of four execution viewpoints to frame our stakeholders concerns. In this section, we discuss the initial definition of the execution viewpoints and describe how we validated three of them.

4.1. Initial definition of execution viewpoints

The findings show that the predefined viewpoints (see Table 1) are useful, but we needed to define specific viewpoints with guidelines to deal with specialized concerns such as managing system complexity and size, making links with other system views explicit, and describing and analyzing actual resource usage. Thus, we defined four specialized viewpoints. Two viewpoints are based on predefined viewpoints (concurrency and deployment) and two are additional viewpoints (functional mapping and resource usage). We presented an extended version of the initial definition in Callo Arias et al. (2009) and in a technical report inside Philips Healthcare MRI. In this section, we focus on the concerns and requirements that motivated the creation of new viewpoints or the customization of the predefined viewpoints.

4.1.1. Functional mapping

Certain practitioners, such as managers and architects are typically more familiar with the functionality and the main functional components of the system. By contrast, designers and platform support engineers are often more familiar with processes and threads. For this reason, a number of practitioners are concerned with how to describe and analyze the relations between the system functionality, system functional components (software components), and actual runtime elements. The functional mapping viewpoint frames this concern and provides guidelines to construct and use functional mapping views.

A functional mapping view is composed by execution models that describe the relations between functional components (interacting together to deliver the system functionality) and actual runtime elements (including software and hardware elements). The main requirement for large and heterogeneous systems is that functional mapping views should enable practitioners who are less familiar with execution elements to understand the actual runtime of the system consistently and without being overwhelmed by the size and complexity of the system. Section 5.1 describes further details and the extension of the definition of this viewpoint.

4.1.2. Execution deployment

This viewpoint is a customization of predefined deployment viewpoints (Clements et al., 2002; Rozanski and Woods, 2005). We defined this customization to support the description of the allocation of system execution elements to processing nodes and the environment into which the system is deployed. Compared

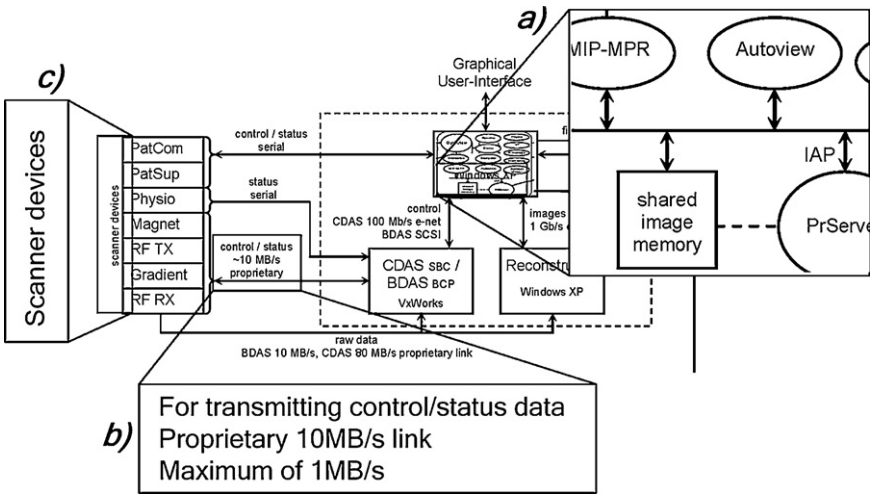


Fig. 5. Customized deployment model for an execution view.

to predefined deployment viewpoints, the requirements that we identified indicate that such a deployment view should show additional information on three aspects (see Fig. 5):

(a) *Detail of processing nodes*: Boxes that describe processing nodes in a deployment model should describe more consistent and useful information. For instance, the predefined deployment viewpoint (Clements et al., 2002), describes that runtime platform and network models should include information about the characteristics of the processing nodes and the functional elements inside them. To do this for a complex system, while keeping an overview, we decided to represent functional elements with software components (groups of processes) thereby reducing complexity when the number of processes is large and details are not necessary. In addition, we identified that it is required to describe the allocation of important code libraries, data repositories, and system-specific hardware devices to processing nodes, making explicit distinctions between these elements and software components.

(b) *Detail of links between processing nodes*: Often deployment models use lines to describe links between processing nodes such as network or communication lines. However, these links often lack descriptions about what they actually serve for at runtime. We identified that for an execution view, links should describe at least three aspects: the function of the link, the link's technology characteristics, and the capacity or bandwidth the system requires from the link.

(c) *Organization of processing nodes*: We identified that the diagrammatic representation of a deployment model should resemble as much as possible the actual physical and geographical distribution of the system. This is particularly required to make some design decision explicit, such as safety issues and rules to manage the influence of physical phenomena (e.g. magnetism) on processing nodes. For instance, the diagram can indicate how processing nodes and the software components they contain can be located close to user interface elements or scanner control devices.

4.1.3. Resource usage

The practitioners we interviewed were also very concerned with the adequate resource usage of the system at runtime. The resource usage viewpoint frames this concern and provides guidelines to construct and use resource usage views. The execution models in a resource usage view describe the metrics, rules, protocols, and budgets that define and govern how the software actually accesses or uses available resources such as data, system code artifacts (software), and runtime platform resources (hardware and software).

It is important to notice that describing resource usage is different from describing required resources. The latter is addressed by the deployment viewpoint, where deployment models describe network connections with the capacity of the physical network link. Instead, resource usage models describe the actual capacity used overtime enabling the analysis of the difference between the required (budgeted) network capacity and the provided capacity. For example, Fig. 6 presents a set of execution models that we constructed to initiate the description and analysis of the actual processor usage of two alternative designs for a key feature of the Philips MRI scanner.

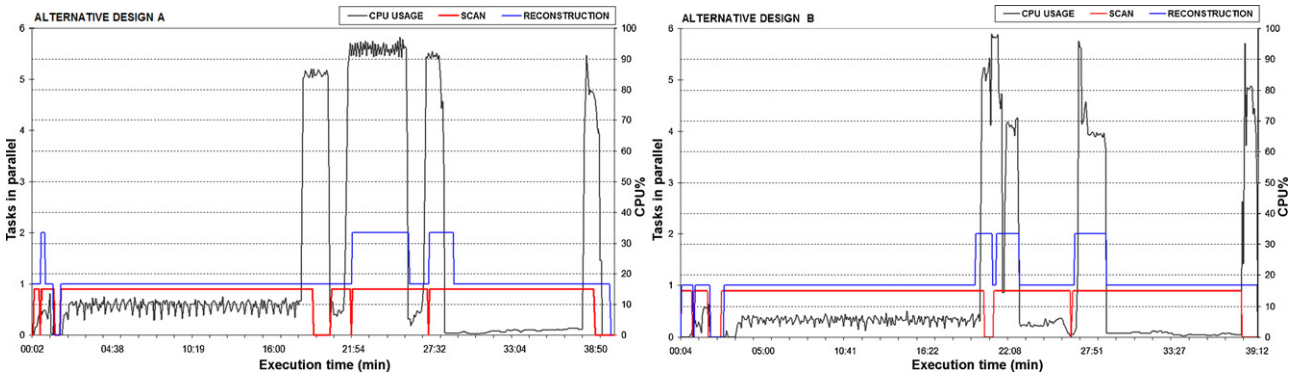


Fig. 6. Examples of describing resource usage to analyze alternative designs.

Table 4
Validation projects for the initial definition of execution viewpoints.

Project	Goal	Practitioners	Viewpoints
1	Redesign of data configuration management for dedicated hardware	Architect and a designer	Functional mapping
2	Tune data-intensive and computation-intensive features	Architect, designer, platform support engineer, and provider	Execution profile Resource usage
3	Improve the system start-up	Architect, designer, platform support engineer, and team leaders	Execution profile Execution concurrency

4.1.4. Execution concurrency

This viewpoint is a customization of the predefined concurrency viewpoint (Rozanski and Woods, 2005). We defined this customization to support the description of actual control flow and data flow between runtime elements. Practitioners are concerned with actual control and data flow because these comprise the runtime behavior of a system in terms of order of interactions, situations of concurrency, communication channels, and time-based interaction. Thus, we identified that it is necessary to describe actual control and data flow but at an overview level, especially to make the dependencies between processes, threads and other system elements (data repositories and the runtime platform elements) explicit.

Reviewing examples of concurrency models, as part of the interviews, we identified that proper abstractions and pragmatic notations are essential to describe the actual concurrency at an overview level. The abstractions that we have identified are data sharing, procedure call, and execution coordination (see Fig. 4). These abstractions should help the characterization and aggregation of actual execution activities between the processes or threads of interacting software components. We observed that pragmatic notations are informal representations that practitioners use guided by practical experience and observation rather than theory. For example, boxes are associated with software components and processing nodes but nothing particular for processes and threads. Therefore, it was necessary distinctive, yet pragmatic, notations for processes and threads, e.g., parallelograms or simple UML diagrams using stereotypes. Section 5.3 describes further details and the extension of the definition of this viewpoint.

4.2. Validation of the initial definition of execution viewpoints

Viewpoints provide a set of reusable guidelines that help architects to construct and effectively use architecture descriptions, organized into the corresponding views. Thus, the value of a viewpoint can be established if the viewpoint proves to be reusable and readily applicable, perhaps after small customizations, across different development projects. To establish this for our initial definition of viewpoints in practice, we were involved in several development projects within Philips Healthcare MRI. Table 4 summarizes three development projects that gave us the opportunity to apply and fine-tune the initial definition of three of the four execution viewpoints. The table includes the goal of the development project, the involved practitioners, and the execution viewpoints used in the projects.

In the projects, the application of the initial definition of viewpoints mainly supported a reverse architecting approach to construct execution views for the Philips Healthcare MRI scanner. The validation of the viewpoints in the three projects took place as follows:

- In the first project, we introduced the initial definition of the functional mapping viewpoint to a software architect and a designer leading the project. The viewpoint supported the construction and presentation of a set of functional mapping models, especially models that describe the relations between the system functionality, functional components, and runtime aggregations of configuration data. These models enabled the top-down analysis and identification of runtime dependencies in the data configuration of dedicated hardware devices in the Philips Healthcare MRI scanner. In this project, we identified the need to analyze and zoom into the details of the relations described by functional mapping models. Together, the descriptions of the mapping relations and their details provided an outline or profile of the analyzed system function or feature. Building on this result, we extended and renamed the initial definition of the functional mapping viewpoint as the execution profile viewpoint (see Section 5.1).
- In the second project, we introduced the definitions of the execution profile and resource usage viewpoints to the same software architect from the first project and a different designer. The definition of the viewpoints guided the construction and use of an execution view of key data-intensive and computation-intensive features of the Philips Healthcare MRI scanner. The goal of the project required the participation of a platform support engineer and a provider of third-party components, with whom the architect and the designer analyzed a set of tradeoffs and the impact of using third-party components in the involved features. In this project, we reused the definition of execution profile viewpoint and extended the definition of the resource usage viewpoint. We identified how to construct resource usage models at different levels of abstraction, e.g., task, component, and process–thread level, and their respective value for various practitioners (see more details in Section 5.2).
- In the third project, we introduced the definitions of the execution profile and execution concurrency viewpoint to the same architect and designer of the second project. The definition of the viewpoints guided the construction and use of an execution view for the start-up of the Philips Healthcare MRI scanner. In contrast to the previous projects, the practitioners combined the construction of the execution view with sketching execution models according to the definitions of the execution viewpoints. It helped to discuss the hypothesis of the actual runtime of the system start-up, but also to perceive the acceptance of the viewpoints by practitioners. The goal of the project required the participation of several team leaders and a platform support engineer to analyze the opportunities of improvements described in the constructed view. In addition, we extended the definition of the execution concurrency viewpoint identifying how to construct execution models that describe concurrency at the level of task and finer workflow entities (see more details in Section 5.3).

5. Documentation of execution viewpoints

The validation of the initial definition of the execution viewpoints helped us to verify and elaborate the identified requirements and concerns for execution views interacting with various practitioners. This allowed us to fine-tune the initial definition of execution viewpoints and construct a more comprehensive documentation of the validated execution viewpoints. In order to align with the ISO/IEC 42010 std. we used the documentation template proposed in the standard for the four execution viewpoints (Callo Arias et al., 2010). From the proposed template (see Fig. 7), the

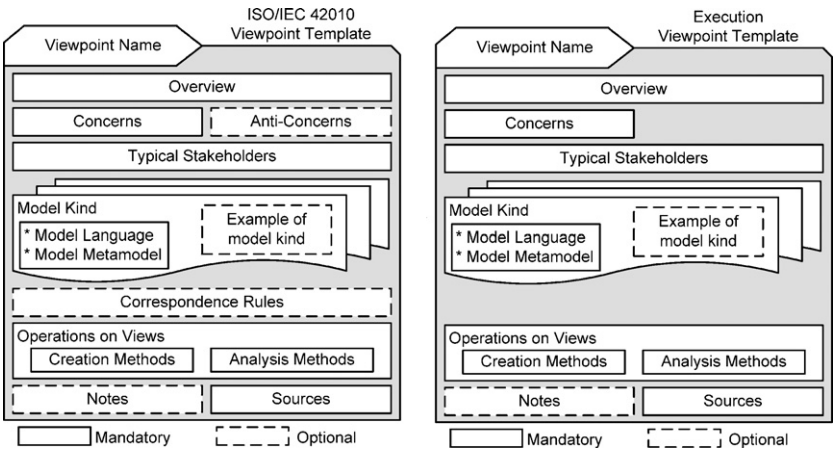


Fig. 7. Templates to document viewpoints.

fields that are included in the description of the next sections are the overview of the documented viewpoint, the set of architectural concerns framed by the viewpoint, the typical stakeholders that hold these concerns, and the kinds of execution models. We have added also two extra fields: construction guidelines, and use guidelines. The construction guidelines describe how to construct the kind of models that address the concerns framed by the respective viewpoint. The use guidelines describe how to use the constructed models to support a set of usual development activities.

5.1. Execution profile (formerly called functional mapping)

The execution profile viewpoint supports the construction and use of an execution profile view. An execution profile view consists of models that provide overview and facilitate the description of details about the runtime of a software-intensive system's functionality, especially without being overwhelmed by the size and complexity of the system implementation.

5.1.1. Concerns and stakeholders

The information described by an execution profile view represents actual and tangible evidence to support top-down analysis activities that address the following concerns:

- What are the major components that realize a given system function?
- What are the high-level dependencies that couple major components?
- What is the development team that develops or maintains a given system's function?

The typical stakeholders that hold these concerns include project leaders, architects, testers, operating system supporters, and newcomers in a development organization.

5.1.2. Model kinds

The kind of models that stakeholders can use to address the concerns framed by this viewpoint include functional mapping and dependency matrix models. These kinds of models support the description of the runtime of a system using high-level elements (e.g., tasks, software components, processes), aggregations that characterize data and code resources, and detailed runtime information.

A functional mapping model is a graph-based representation that describes relationships between high-level elements of a key execution scenario. Fig. 8 illustrates an example of a functional mapping model. The notations of a functional mapping model consist of four aspects. (1) A scenario is described as a set of tasks linked to the software components that realize each of them, e.g., using color-coded edges. (2) Each software component is described together with its corresponding set of running processes, e.g., using a record structure whose fields represent the processes, following the definition of a software component described in Section 3.3.2 and illustrated in Fig. 4. (3) The links from the task to the software components continue to describe the software components' runtime activity, e.g., read, write, and execute on the involved data, code, or platform resources of the system. (4) The involved data, code, or platform resources are represented as high-level aggregations. For example, Configuration Repository, in Fig. 8, aggregates a set of configuration files that are used in the given scenario.

A dependency matrix model is a matrix-based representation that supports the analysis of relationships and their details to determinate dependencies between major runtime elements. Fig. 9 illustrates a dependency matrix model. The notations of this matrix model include the following two aspects: (1) Rows and columns represent high-level abstractions (tasks or software components); (2) The cells in a dependency matrix represent quantifications of the elements that build the high-level abstractions or the runtime activities of these elements (e.g., reading and writing operations). The quantifications in the cells are used to analyze relationships between tasks, software components, or combinations of these elements interacting in the given execution scenario. For example, the matrix in Fig. 9 was constructed to describe relationships between the tasks. The quantifications in the cells are the number of libraries shared by the software components interacting within the given tasks. The sort of information to be described by the columns, rows, and cells can be configured, based on the provided tool support and the stakeholders' concerns.

5.1.3. Guidelines to construct an execution profile view

- Given the system functionality under analysis, the architect needs to choose a set of key execution scenarios (e.g., test cases and integration tests), which cover the representative system's runtime behavior and structure related to the given system functionality.
- To manage size and complexity, a key scenario should be decomposed into a sequence of tasks. A task of an execution scenario is an aggregation of a set of activities and events triggered by the end-user or automatically executed by the system within the workflow of the scenario.
- Architects can extract information about the actual sequence of tasks or workflow of a scenario from design documents, or runtime data produced by logging mechanisms that are part of the system infrastructure or monitoring utilities provided by the system platform.

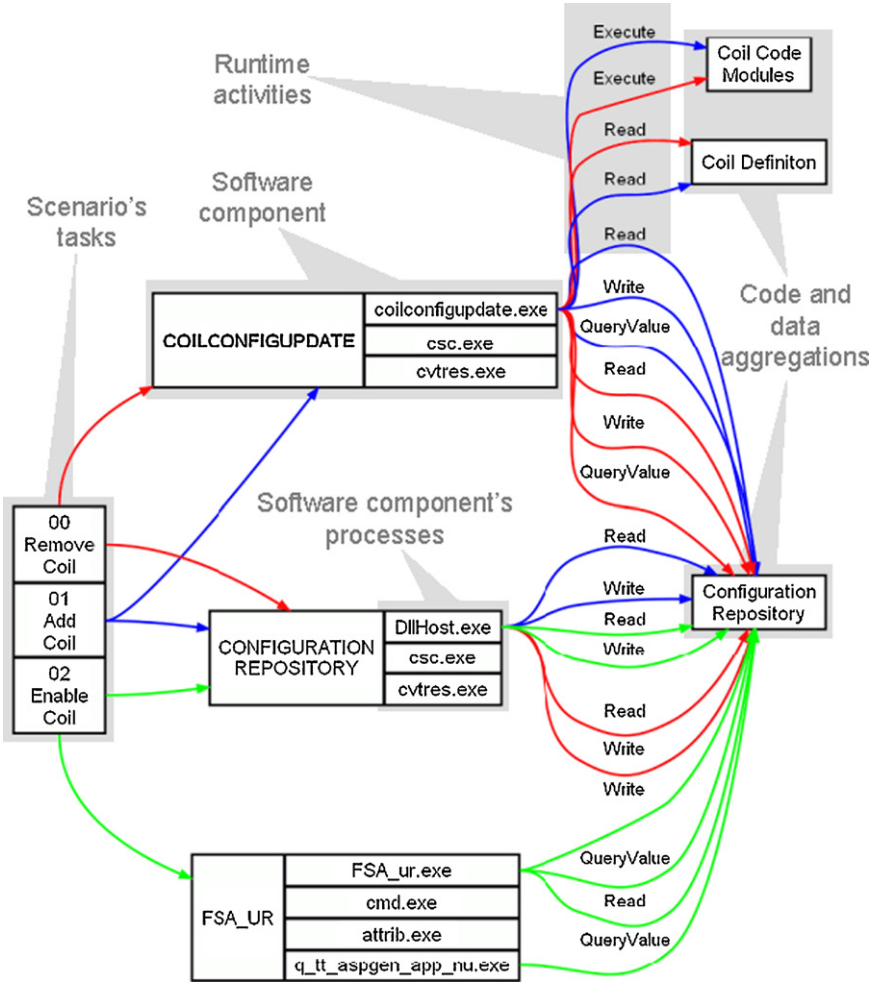
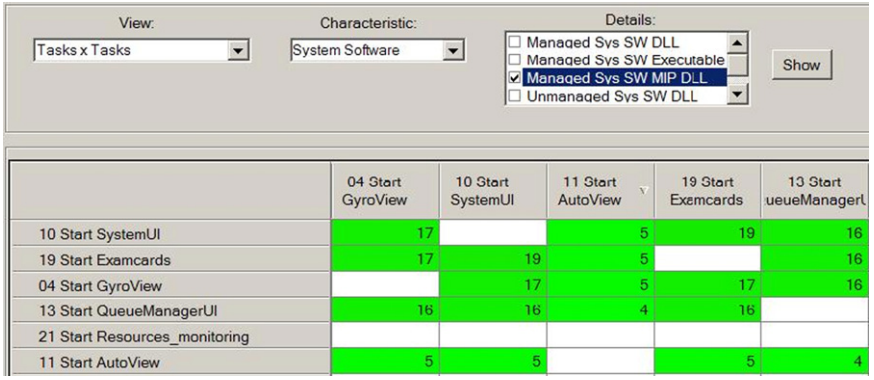


Fig. 8. Example of a functional mapping model.

- When using runtime data to construct an execution profile view, architects should extract the following information: (1) The actual set of involved software components and their corresponding set of processes and threads; (2) Aggregations that represent system data repositories, code libraries/packages, and if possible the system specific hardware devices; (3) The execution activity that describes how software components' processes use data repositories, code libraries, and system specific devices.

5.1.4. Guidelines to use an execution profile view

Both, functional mapping and dependency matrix models describe runtime information about the sequence of tasks within a execution scenario, the set of software components and their respective set of processes, and the distinction of the execution activity per task and per software component on data repositories, code modules, and system-specific resources. This information helps the various stakeholders to analyze the runtime of a system in the following ways:



	04 Start GyroView	10 Start SystemUI	11 Start AutoView	19 Start Examcards	13 Start ueueManagerU
10 Start SystemUI	17		5	19	16
19 Start Examcards	17	19	5		16
04 Start GyroView		17	5	17	16
13 Start QueueManagerUI	16	16	4	16	
21 Start Resources_monitoring					
11 Start AutoView	5	5		5	4

Fig. 9. Example of a dependency matrix model.

- Project leaders and newcomers can use an execution profile view to learn about the system functionality, the set of major components (hardware, software, and data) that realize it, and the high-level dependencies that couple them.
- Execution profile views contain information to support downstream planning of development projects. For instance, stakeholders can identify the development force, i.e. internal and external teams that are in charge of the development and maintenance of the identified components that perform a given system function to be changed within a development project.
- For testers and operating system supporters, an execution profile view provides information to identify the actual processes and execution elements such as data repositories and platform resources that may influence or play a role in the design of test cases, the assessment of test results, and the report for corrective maintenance activities.
- Architects and project leaders can use dependency matrix models to identify relationships between the tasks of a scenario, between the software components of a scenario, and between the tasks and software components of a scenario. Characterizing an identified relationship as dependency will respond on the impact of change perceived by the stakeholder.

5.2. Resource usage viewpoint

Software intensive-systems include software and hardware elements. Software elements are considered as sets of instructions that govern the use of hardware elements (Woodside, 2001), such as processors, memory, disk, and network interfaces. The resource usage viewpoint supports the construction and use of a resource usage view. A resource usage view consists of models that provide overviews and facilitate the description of details of how the software elements of a software-intensive system use hardware elements at runtime.

5.2.1. Concerns and stakeholders

The information described by a resource usage view addresses the following concerns:

- 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 govern the use of hardware resources at runtime?
- How do the various types of software elements (e.g., proprietary and third party) consume resources such as processor and memory within key execution scenarios?
- Does the realization of the system implementation has an efficient resource usage?
- What are the bottlenecks and delays of the system and their root cause?

The typical stakeholders that hold these concerns include system administrators, platform/infrastructure supporters, architects, designers, software engineers, and testers.

5.2.2. Model kinds

A resource usage view includes models that stakeholders can use to describe the as-is usage of hardware resources (e.g., processor, memory, and network) within a given execution scenario at different levels of abstraction. According to the level of abstraction, resource usage models can be classified as three kinds of models: task, component, and process–thread resource usage models.

The first model kind, *task resource usage models*, is the most coarse-grained representation of resource usage information. A model of this kind describes resource usage showing the correlation between the duration of the tasks of an execution scenario and the consumption of the given hardware resource(s). For example, the model in Fig. 10 describes memory usage across the tasks of the Philips MRI scanner's start-up. The second model kind, *component resource usage models*, is a finer representation of resource usage information. A model of this kind describes resource usage showing the correlation between the duration of software components' runtime activity and consumption of the given hardware resource(s). Fig. 11 illustrates an example of this kind of model that describes the processor usage of the software components in the Philips MRI scanner's Recon computer, which serves the main computation-intensive function of this system. The third model kind, *process–thread resource usage models*, is the most fine-grained representations of resource usage. A model of this kind describes resource usage showing the correlation between the duration of processes and threads' runtime activity and the consumption of the given hardware resource(s). Fig. 12 illustrates an example of this kind of model.

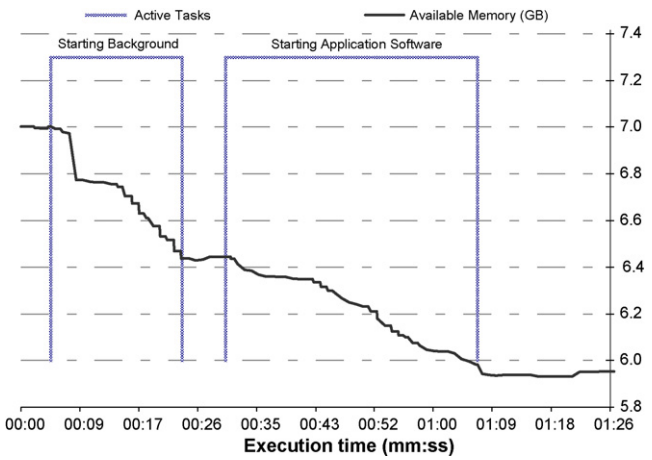


Fig. 10. Example of a task resource usage model.

The three kinds of resource usage models share the following four common notations. (1) Horizontal bars represent aggregations of runtime activity at task, component, or thread level. The length of a horizontal bar represents the duration of the aggregated runtime activity. (2) Aggregations are distributed along a horizontal time axis to illustrate their occurrence over time. (3) Aggregations should be vertically distributed to assemble their actual distribution onto the system processing nodes. For example, Fig. 11 illustrates aggregations of runtime activity at the component level, which are vertically distributed across two system computers, Scanner and Recon. (4) Each kind of model can require two vertical axes. A left axis is the reference to identify the involved runtime element, e.g., software components or thread. The right axis is the reference for the metrics of the resource usage values, e.g., gigabytes of consumed memory.

5.2.3. Guidelines to construct a resource usage view

- Resource usage descriptions should be based on actual resource usage measurements which can be collected using tools such as Process monitor or Windows Performance Analyzer (Russinovich, 2010).
- Runtime measurements should be collected from a set of execution scenarios, which the development organization identifies as a representative benchmark of the system feature under analysis.
- The benchmark should be run using a representative input, e.g., data sets, to capture a representative behavior of the hardware resources involved in the feature under analysis.

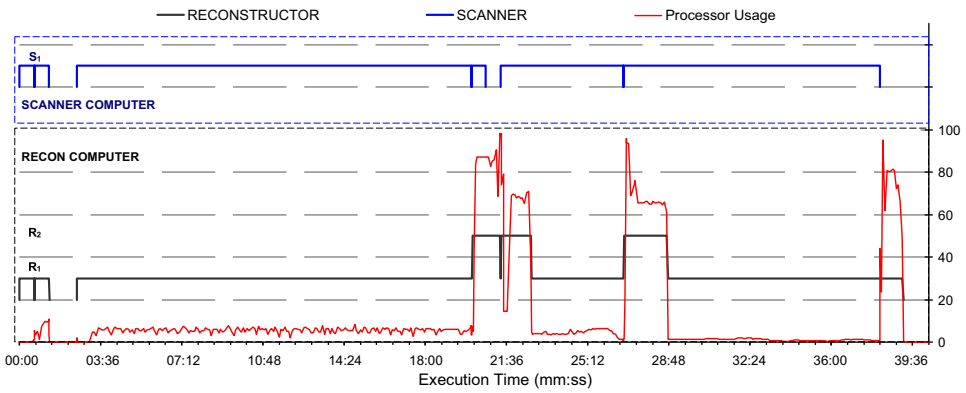


Fig. 11. Example of a component resource usage model.

- To correlate resource usage information with architectural abstractions, runtime measurements should be complemented with workflow information (extracted form sources such as logging). Thus for descriptions at the software component level e.g., Fig. 11, runtime information should include: (a) the actual set of involved software components and their corresponding set of processes; (b) the execution periods of each software component that is involved in the execution scenario. For descriptions at the thread level, e.g., Fig. 12, runtime information should include the actual set of involved process and their respective threads.
- To identify the set of actual threads, it will be useful to have at hand a concurrency model of the scenario under analysis (see Section 5.3).
- In addition, execution information should include the execution periods of the identified threads, i.e. aggregations of consecutive thread execution activity, and when possible the control and dataflow between threads.

5.2.4. Guidelines to use a resource usage view

The information described by resource usage models help the identified stakeholders to address their concerns about the resource usage in the following ways:

- Software architects, designers, and platform supporters, can use task resource usage models to identify, predict, and tune resource usage budgets. For example, the model in Fig. 10 helped architects to identify the actual memory usage across two of the main task of the start-up of the Philips MRI scanner.
- Software designers may also use resource usage models, e.g. Fig. 11, to analyze alternative architectures or designs and compare them based on how efficiently processors or memory are used to deliver key computation- or data-intensive system functions.
- Designers and software engineers can use resource usage models to identify opportunities to tune and match design and implementation. For instance, models like Figs. 11 and 12 helped to identify correlations between delays or dead times analyzing peaks and valleys in the representation of resource usage activity.
- In overall, resource usage models are useful evidences that ease the communication and sharing of knowledge between internal and external teams. For example, designers, platform support engineers, and external providers can drill down into the actual resource usage of a component, process, or thread without looking at the implementation code.
- Testers may use resource usage models in the definition and improvement of benchmarks for the design and execution of test and verification procedures. Having resource usage models of a given execution scenario before and after it is changed, serve as evidence to track, describe, and communicate the desired or undesired variations of the runtime of the system.

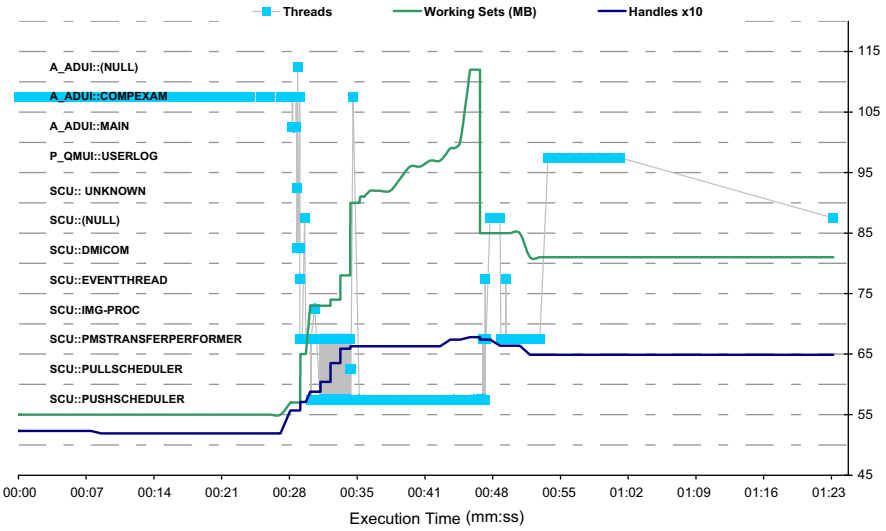


Fig. 12. Example of a process–thread resource usage model.

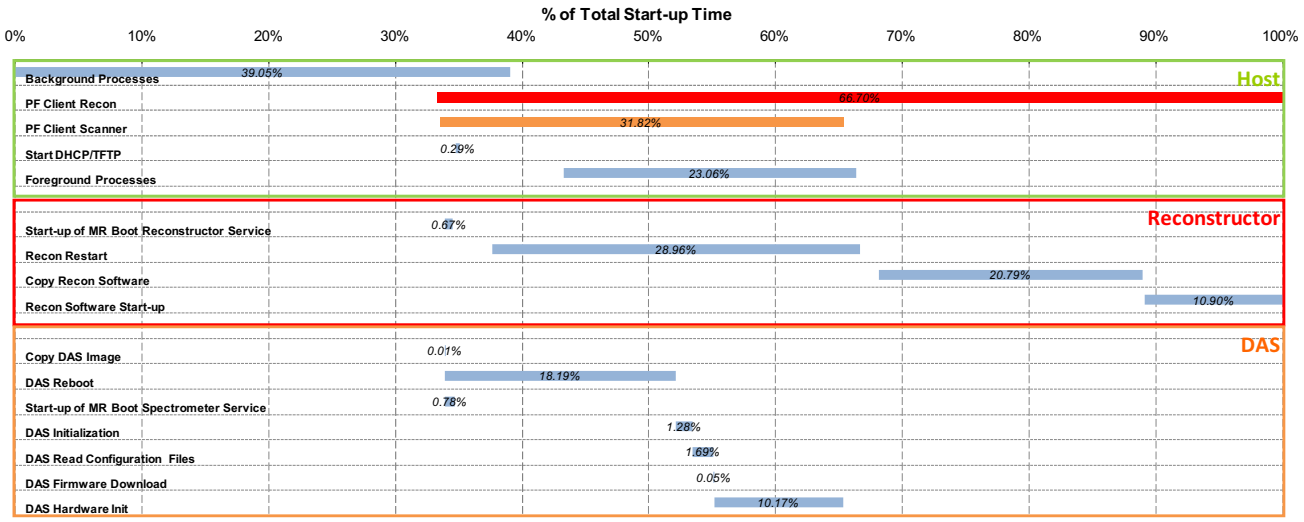


Fig. 13. Example of a workflow concurrency model.

5.3. Execution concurrency viewpoint

The execution concurrency viewpoint supports the construction and use of execution concurrency views. An execution concurrency view consists of models that provide overviews of how the runtime elements of a software-intensive system execute concurrently at runtime. The execution concurrency viewpoint is a customization of the predefined concurrency viewpoint (Rozanski and Woods, 2005). We identified that in practice the runtime concurrency of a system often deviates from its designed concurrency, which implies differences between the designed and the actual control flow and data flow between software components. Control flow defines the order of execution and synchronization between software components to use or access the various system resources. Data flow describes how data is processed and flows through software components and other system elements such as data repositories.

5.3.1. Concerns and stakeholders

An execution concurrency viewpoint frames the following concerns:

- Which runtime elements execute concurrently?
- How does the runtime concurrency match the designed concurrency?
- What are the aspects that constrain, coordinate, and control the system's runtime concurrency?
- What are the opportunities to improve the system's runtime concurrency?

These concerns are hold by stakeholders like architects, designers, software engineers, testers, and operating system supporters.

5.3.2. Model kinds

The kinds of models that address the concerns framed by an execution concurrency viewpoint include workflow concurrency and process–thread structure models.

A workflow concurrency model is a Gantt-chart like representation that illustrates temporal relations between high-level runtime elements (e.g., tasks or software components). Fig. 13 presents an example of a workflow concurrency model. This model describes the runtime concurrency of tasks that make up the complete start-up of the Philips MRI scanner. The notation of models of this kind includes the following three characteristics: (1) Elements such as scenario's tasks are represented as horizontal bars. (2) The horizontal organization of these bars corresponds to a time axis, which is the reference to describe the duration of a task over time. (3) The vertical organization of a task describes its distribution across the involved processing nodes. Color-coding can be useful to distinguish the function or nature of the involved tasks and the borders between the containing processing nodes.

A process–thread structure model describes the distribution and mapping of functional elements to runtime platform elements such as processes and their threads. Fig. 14 presents an example of a process–thread structure model, which describes the process and thread structure of a data-intensive feature of the Philips MRI system. The information described in the model includes an instance of the actual control- and data-flow for the execution scenario under analysis. This model complements the resource usage model presented in Fig. 12. The notation of models of this kind includes the following three characteristics: (1) Runtime processes are rep-

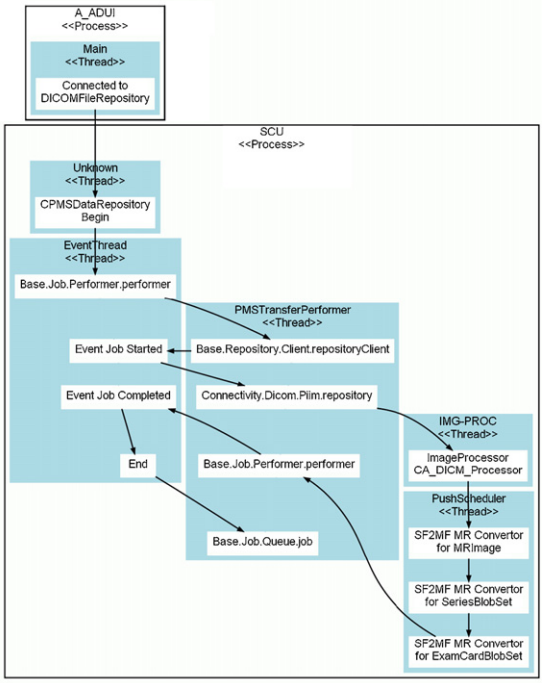


Fig. 14. Example of a process–thread structure model.

resented as containers of threads. At the same time, threads are represented as containers of code modules, runtime events, and interfaces to data and hardware resources. (2) The notations of containers can be usual boxes, and lines connecting them as representation of control and data flow relationships. Richer notations such as UML and stereotyping are other alternatives. For example, the containers in the model of Fig. 14 are represented using stereotypes to distinguish processes from threads.

5.3.3. Guidelines to construct an execution concurrency view

- Runtime concurrency information should be based on actual runtime information which can be extracted from runtime data collected using tools such as Process monitor (Russinovich, 2010) or the system's logging mechanisms.
- The development organization should choose a set of execution scenarios (e.g., test cases and integration tests) that are representative for the system functionality to be described.
- To construct concurrency workflow models, the architect has to identify the important tasks that build the chosen scenarios. The identification of the tasks includes the identification of the start time and duration of each task. In addition, the distribution of the tasks should be also identified, preferably from runtime data though design knowledge can be useful as well.
- To construct process–thread structure models, the architect has to identify the important runtime processes involved in the execution scenario under analysis. This is especially necessary for systems with large and complex runtime process and thread structures. To do so, the architect can analyze runtime data using design knowledge to filter out less important process and threads.
- When using runtime data, it is important that tasks, processes, and threads are identified with meaningful names rather than numeric identifications. This is important to match runtime information to system design information.

5.3.4. Guidelines to use a execution concurrency view

The information described by workflow concurrency and process–structure models help stakeholders to analyze the runtime concurrency of a system in the following ways:

- Architects and designers may use concurrency workflow models to gather high-level information about elements that run concurrently as input for the downstream planning of development activities.
- Testers can use concurrency model in the definition, design, and execution of test and verification procedures. For example, testers can use concurrency workflow models as evidence to track and communicate the desired or undesired variations of the runtime concurrency of the system.
- Software engineer can use concurrency models to learn and analyze how the pieces of code they implement are instantiated and deployed at runtime.
- Stakeholders concerned about resource usage can use process–thread structure models to understand the runtime structure that governs a given resource usage. For example, the model in Fig. 14 supports or complements the one in Fig. 12.
- In overall, execution concurrency models are useful to share and communicate technical knowledge between operating system, platform supporters, and architects and designers.

6. Conclusions

We described how to define, validate, and document a set of execution viewpoints to support the construction and use of execution views for an existing large software-intensive system based on the requirements of its development organization. The contribution of our approach is three-fold. First, we have shown and conceptualized how to use (customize and extend) predefined viewpoints in practice. Second, the definition approach using predefined viewpoints is a valuable complement (e.g., to scope and guide) to more general-purpose definition methods such as (Koning and van Vliet, 2006). Moreover, our approach is repeatable in other organizations and research groups. The practitioners involved in the phases of the approach confirmed that a similar approach could be used to upgrade or define other viewpoints for their specific system. Third, other development organization and researchers can reuse and extent the documented execution viewpoints to support the construction and use of execution views for other systems. This is specially recognized by the new CD2 version of the ISO/IEC 42010 std., which referees to our initial definition of the approach (Callo Arias et al., 2009) as a representative example of how to define viewpoints.

Acknowledgments

We would like to thank the Software Architecture Team and the software designers of the MRI scanner in Philips Healthcare, in particular Krelis Blom and Danny Havenith. We also extend our gratitude to Rob van Ommering, Wim van der Linden, and our Darwin colleagues for their feedback and joint work.

This work has been carried out as a part of the Darwin project at Philips Healthcare under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

Appendix I. Example of a model-specific questionnaire

AD Project name: Building the Execution Architecture of the MRI System				**Date:**
Domain:	**Team:**			
Activity: Review of Execution Architecture Documentation				
Purpose of the activity:				
Review Session: Runtime Structure or Concurrency Models In this session, we review in detail the section Runtime structure of the document Execution Architecture and the concurrency or behavior viewpoints from the literature. The review is centered in discussing in detail the concerns addressed by this section and some of the diagrams of the runtime structure of the MRI system execution.				
1. Creation and maintenance overview: - **Is there any specific contributor or source of information?** - **Besides the guidelines of the 4+1 model, what triggered the creation of this section?** - **What was the validation of the information of this section?** - **How often is this section going to change?**				
2. Intended audience: (roles*) Hardware and Software designers and architects			**3. Actual audience: (roles*)** * Roles within PH MRI e.g. architect, designer, implementer, maintainer, etc.	
4. Usage w.r.t. architecting and design activities The tailoring of the list of activities is based on the overview review (previous session)				
Activity	**Intended**	**Actual**	**Desired**	**Comments and brief answers on how the activity is addressed**
Communication among development units				
Conformance of downstream design and development				
Analysis & Design workflow				
Education and training				
Communication with customers and/or providers				
Analysis of system quality attributes				
Analysis of alternative architectures/designs				
Other specific activities for an improved version of this section				
Planning and creation of vision and roadmaps				
5. Usage w.r.t. specific (architectural and design) concerns addressed by a concurrency view point Concerns are collected from the literature, nevertheless we expect that the interviewee may add some specific concerns				
Concern	**Intended**	**Actual**	**Desired**	**Comments or brief answers on how the concern is addressed**
Process/Thread Structure				
Show the mapping of functional elements to Process/Thread(s)				
Describe the mapping of functional elements to Process				
Explain the mapping of functional elements to Process				
Inter-process communication (Which are/why)				
State management (states, transitions, causes, and effects)				
Synchronization and integrity (e.g. mutex and shared data)				
Startup and shutdown of unit and the aggregate system				
Failure (Thread level and process crash) and propagation				
Reentrancy and priorities (critical sections, shared code)				
Notes:				
6. Description and representation of information (in the provided runtime views: Figure 1 and Figure 2)				
Question	**Possible alternatives**			**Comments and brief answers**
What is the a bstraction level of the diagram?	System	Overview	Detail	
Do you recognize the type or class of elements described by edges and nodes?				
Do you recognize interactions between elements?				
Do you understand what happened due to interactions?				
Do you identify the sequence of interactions				
Do you recognize what is inside of the nodes?				
Can you describe the reason for grouping elements inside nodes?				
Can you recognize the semantic of the different edges?				
Additional Comments				
• *Attached models (System level, Overview level, Detail level)*				

References

Callo Arias, T.B., Avgeriou, P., America, P., 2008. Analyzing the actual execution of a large software-intensive system for determining dependencies. In: Presented at 15th Working Conference on Reverse Engineering.

Callo Arias, T.B., America, P., Avgeriou, P., 2009. Defining execution viewpoints for a large and complex software-intensive system. In: Presented at Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture.

Callo Arias, T.B., Avgeriou, A., America, P., March 2010. Tech. Report: Documenting a Catalog of Viewpoints to Describe the Execution Architecture of a Large Software-Intensive System for the ISO/IEC 42010 Std., http://www.esi.nl/projects/darwin/publications/.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures: Views and Beyond. Addison Wesley.

Hofmeister, C., Nord, R., Soni, D., 1999. Applied Software Architecture. Addison-Wesley.

Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P., 2007. A general model of software architecture design derived from five industrial approaches. Journal of Systems and Software 80, 106–126.

Hopkins, R., Jenkins, K., 2008. Eating the IT Elephant: Moving from Greenfield Development to Brownfield. IBM Press.

Hunt, G., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., Zill, B., 2007. Sealing OS processes to improve dependability and safety. In: 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems. ACM.

ISO/IEC-JTC1/SC7, ISO/IEC 42010, 2007. Systems and Software Engineering—Recommended Practice for Architectural Description of Software-Intensive Systems.

Koning, H., van Vliet, H., 2006. A method for defining IEEE Std 1471 viewpoints. The Journal of Systems & Software 79, 120–131.

Kruchten, P., 1995. The 4+1 View Model of architecture. IEEE Software 12, 42–50.

Muller, G., 2004. CAFCR: a multi-view method for embedded systems architecting; balancing genericity and specificity. Ph.D. Thesis, Technical University Delft, The Netherlands.

Muller, G., April 2009. Gaudí System Architecting—A Reference Architecture Primer., http://www.gaudisite.nl/info/ReferenceArchitecturePrimer.info.html.

Obbink, H., Kruchten, P., Kozaczynski, W., Hilliard, R., Ran, A., Postema, H., Lutz, D., Kazman, R., Tracz, W., Kahane, E., November 2008. Report on Software Architecture Review and Assessment Version 1. 0., http://philippe.kruchten.com/architecture/SARAv1.pdf.

Philips Healthcare: Magnetic Resonance Imaging, March 2010. http://www.healthcare.philips.com/main/products/mri.

Rozanski, N., Woods, E., 2005. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison Wesley.

Russinovich, M., March 2010. The Sysinternals Utilities., http://technet.microsoft.com/en-us/sysinternals/.

Sozer, H., Tekinerdogan, B., 2008. Introducing recovery style for modeling and analyzing system recovery. In: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE Computer Society.

van de Laar, P., America, P., Rutgers, J., van Loo, S., Muller, G., Punter, T., Watts, D., 2007. The Darwin Project: Evolvability of Software-Intensive Systems. In: Presented at Third International IEEE Workshop on Software Evolvability.

van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C., 2004. Symphony: view-driven software architecture reconstruction. In: Presented at IEEE/IFIP Working Conference on Software Architecture.

Woodside, C., 2001. Software resource architecture and performance evaluation of software architectures. In: Presented at 34th Annual Hawaii International Conference on System Sciences.

Trosky B. Callo Arias received an Engineer's degree in informatics and systems from Universidad Nacional San Antonio Abad del Cusco-Peru in 2002, and a Master's degree in computer science from Göteborg University-Sweden in 2005. He is a Ph.D candidate in the Software Engineering and Architecture Group of University of Groningen. His professional interest includes the architecture and design of software solutions for high-tech products, embedded systems, and distributed systems.

Pierre America received a Master's degree from the University of Utrecht in 1982 and a Ph.D. from the Free University of Amsterdam in 1989. He joined Philips Research in 1982 where he has been working in different areas of computer science, ranging from formal aspects of parallel object-oriented programming to music processing. During the last years he has been working on software and system architecting approaches for product families. He has been applying and validating these approaches in close cooperation with Philips Healthcare. Starting in 2008 he is working part of his time as a Research Fellow at the Embedded Systems Institute, where his main focus is on evolvability.

Paris Avgeriou is Professor of Software Engineering in the Department of Mathematics and Computing Science, University of Groningen, the Netherlands where he has led the Software Engineering research group since September 2006. Before joining Groningen, he was a post-doctoral Fellow of the European Re-search Consortium for Informatics and Mathematics (ERCIM). He has participated in a number of national and European research projects directly related to the European industry of Software-intensive systems. He has co-organized several international workshops, mainly at the International Conference on Software Engineering (ICSE). He sits on the editorial board of Springer Transactions on Pattern Languages of Programming. He has published more than 90 peer-reviewed articles in international journals, conference proceedings and books. His research interests lie in the area of software architecture, with strong emphasis on architecture modeling, knowledge, evolution and patterns.