A practice-driven systematic review of dependency analysis solutions

**Trosky B. Callo Arias · Pieter van der Spek ·
Paris Avgeriou**

Abstract When following architecture-driven strategies to develop large software-intensive systems, the analysis of the dependencies is not an easy task. In this paper, we report a systematic literature review on dependency analysis solutions. Dependency analysis concerns making dependencies due to interconnections between programs or system components explicit. The review is practice-driven because its research questions, execution, and reporting were influenced by the practice of a group of software architects at Philips Healthcare MRI. The review results in an overview and assessment of the state-of-the-art and applicability of dependency analysis. The overview provides insights about definitions related to dependency analysis, the sort of development activities that need dependency analysis, and the classification and description of a number of dependency analysis solutions. The contribution of this paper is for both practitioners and researchers. They can take it as a reference to learn about dependency analysis, match their own practice to the presented results, and to build similar overviews of other techniques and methods for other domains or types of systems.

T. B. Callo Arias (✉) · P. Avgeriou
University of Groningen, Nijenborgh 9, 9747 AG Groningen, The Netherlands
e-mail: trosky@cs.rug.nl

P. Avgeriou
e-mail: paris@cs.rug.nl

P. van der Spek
VU University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
e-mail: pvdspek@cs.vu.nl

 Springer

1 Introduction

The development of methods and techniques to understand and analyze software systems is an active research area with considerable attention from the software industry. Software organizations are aware of the fact that without sufficient understanding of the systems they develop, maintenance and evolution becomes expensive and unpredictable. For instance, one of the major challenges in software maintenance is the need to determine the effects of modifications made to a program (Loyall and Mathisen 1993). The overall cost of a small change (affecting only a handful of lines of code) can already be extremely high, especially when the information about the interconnections between the components that make up the system is limited or not reliable. This is true even for well-structured systems that minimize but do not eliminate the interconnections among system objects that lead to unexpected effects (Moriconi and Winkler 1990) and dependencies.

Some of the methods and techniques to increase the understanding of software systems are especially geared to conduct dependency analysis. Dependency analysis concerns making dependencies due to interconnections between programs or system components explicit. Over the last decades, researchers have produced a number of solutions (methods, tools, and techniques) to support the analysis of dependencies in software systems. Our interest in dependency analysis has its origin in the context of our research project (van de Laar et al. 2007). We investigate how to improve the evolvability (the ability to respond effectively to change) of software-intensive systems studying a Magnetic Resonance Imaging (MRI) system developed by our industrial partner, Philips Healthcare.

In the context of our project, one of our early observations was that indeed information about dependencies was needed and deemed important to improve the evolvability of the Philips MRI system. However, we also observed that in the development of this system, dependency analysis was a time-consuming activity conducted on an ad hoc basis. This is mainly caused by the size and complexity of the Philips MRI system and the lack of proper support within the organization (e.g., tools and techniques). Even though this situation was a good candidate for improvements, our ability to propose or develop improvements was limited by the fact that practitioners had an unclear perception about the value of dependency analysis and how it could improve their practice. Thus, we decided to build an overview that practitioners could use to improve their knowledge and perception about dependency analysis.

In this paper, we report a systematic literature review (Kitchenham 2004a) that we conducted to build the overview of dependency analysis. The review is practice-driven due to three main factors. First, the research questions come from observations that we collected interacting with a group of architects and designers at Philips Healthcare MRI. Second, the design and execution of the review protocol aimed at finding and presenting research results that can be used by practitioners, rather than research trends upon which researchers can base their future research. Third, the review includes an assessment of research results taking into account the practical characteristics of the Philips MRI system and its architecture-driven development process.

We used the constructed overview to improve the practitioners' knowledge about dependency analysis, and identify the opportunities and constraints to improve

dependency analysis in the practice of our industrial partner. The contribution of this paper focuses primarily in supporting practitioners to learn about the state-of-art in dependency analysis and how it matches the characteristics and development of a representative large and complex software-intensive system. In addition, we consider that other researchers can replicate our study to create similar overviews of dependency analysis (or other techniques and method) for other domains or types of systems.

Structure of the Paper   The remainder of this paper is organized as follows. Section 2 describes the context in which we performed the review and the specific research questions. Section 3 describes the protocol of the review. Section 4 starts the overview describing conceptual aspects related to dependency analysis. Next, Section 5 describes the application areas that dependency analysis contribute to. Section 6 describes a set of existing dependency analysis solutions classified by their source of information. Section 7 completes the overview describing how existing definitions and solutions match the practical requirements of our particular context and the identified opportunities for improvement. In Section 8, we discuss threats to validity for the review and the results. Finally, Section 9 provides some concluding remarks.

2 Context and Research Questions

We conducted the review as part of our research in the Darwin project (van de Laar et al. 2007). This applied research project is currently underway at Philips Healthcare. In this project we focus on how to improve the evolvability of software-intensive systems studying a Magnetic Resonance Imaging (MRI) system. Thus, when mentioning *practitioners* or *in practice* we refer to the developers and the intrinsic development of the Philips MRI system respectively. In the rest of this section, we describe the characteristics of the context of our research project and the research questions that triggered our systematic literature review.

2.1 Magnetic Resonance Imaging Systems

Figure 1a shows an example of a modern MRI scanner. In clinical practice, MRI is used to distinguish pathological tissue (such as a brain tumor) from normal tissue. One advantage of an MRI scan is that it is believed to be harmless to the patient. It uses strong magnetic fields and non-ionizing radiation in the radio frequency range, unlike CT scans (where CT stands for computed tomography) and traditional X-rays which both use ionizing radiation.

MRI scanners are among the most expensive medical equipment available today. High-end scanners, such as the one shown in Fig. 1a, cost around $2.5 million USD. Installation of the system in the hospital costs another $500,000 USD. The MRI system is made up of three main components: a superconducting magnet, a gradient system, and a radio frequency system. The magnet is the largest and most expensive component of the scanner. Its strength is measured in tesla (symbol: T). The high-end scanner shown here contains a 3 T magnet. In comparison, the strength of the Earth's magnetic field on the equator is 31 μT (3.1×10^{-5} T) and the strength of a typical refrigerator magnet is 5 mT.
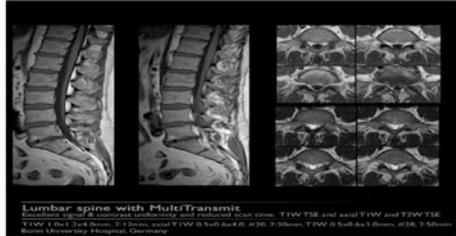
(a) Philips 3T Achieva system. The small open-ing (called the bore) is clearly visible. The casing around it holds the superconducting magnet as well as the liquid helium to cool it.
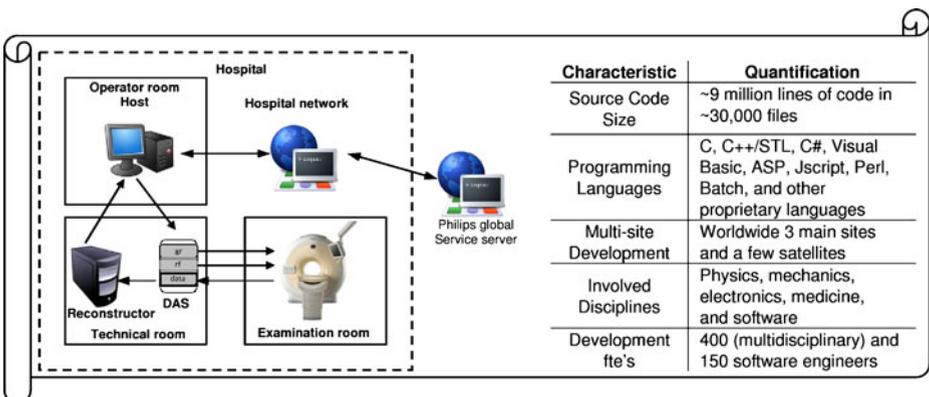
(b) The result of a scan showing a patient's lumbar spine.

Fig. 1 Impression of the Philips Healthcare MRI system

The whole system is controlled by a software system that runs in one or more com-puters. The system with all its related components itself does not fit in a single room (see Fig. 2), but comprises several rooms with different conditional requirements, to house the subsystem components. Additional workstations can be part of the system to allow physician access to the raw image data collected from various scans. Using advanced algorithms, this data is processed into images like the one in Fig. 1b, which allow physicians to diagnose patients during neural examinations.

2.2 The MRI Software

We consider the MRI system as a representative large software-intensive system. It combines various hardware components with a fair amount of software into a single, complex system. Figure 2 provides an impression of the system and organizational complexity involved with, both in the system as well as in the organization. From the previous section, the complexity of the hardware involved is obvious. Various

Characteristic	Quantification
Source Code Size	~9 million lines of code in ~30,000 files
Programming Languages	C, C++/STL, C#, Visual Basic, ASP, Jscript, Perl, Batch, and other proprietary languages
Multi-site Development	Worldwide 3 main sites and a few satellites
Involved Disciplines	Physics, mechanics, electronics, medicine, and software
Development fte's	400 (multidisciplinary) and 150 software engineers

Fig. 2 Overview of the Philips MRI system size and complexity

advanced pieces of hardware are combined into a single product controlled by a proprietary software stack. The software comprises several million lines of code written in nine different programming languages (heterogeneous implementation). Also, the software has a long history of being exposed to numerous changes and is composed of legacy parts associated to large investments in both time and money.

Next to the technical complexity of the system, development of a large software-intensive system also involves a high level of organizational complexity. Figure 2 shows the number of people involved with the evolutionary development of the MRI scanner. MRI development requires a multidisciplinary design team with competences in areas such as physics, electronics, mechanics, material science, software engineering, and clinical science. All of the different disciplines have to work together effectively on many aspects of system design. Adding to the complexity is that the work is carried out in geographically spread locations and in different timezones.

Furthermore, the knowledge about the system is spread among the experts of the organization, and when it comes to the oldest parts of its implementation, this knowledge may be limited because the documentation is either not up-to-date or not readily available. Thus, this system creates several special requirements for the dependency analysis, for instance due to the size of its implementation and the need to cross the barriers imposed by its heterogeneity and complexity.

Finally, the architecture-driven development process at Philips Healthcare MRI is also an important factor. At Philips Healthcare MRI the overall software development is monitored by the Software Architecture Team (SWAT). This team is responsible for the general architecture of the system. The system is decomposed in several subsystems and components which are the responsibilities of software designers. Most of the implementation is done by programmers. The different roles also point to different requirements in terms of the knowledge they require. The architects are mostly interested in high-level, architectural dependencies. On the other hand, programmers see dependencies in terms of source-code level constructs such as function calls. As we will see later, these different perspectives influence various aspects of this review such as the relevant sources of information.

2.3 Research Questions

Table 1 shows the set of research questions that we aimed to answer by conducting the systematic review. We have defined this set of questions trying to generalize the following observations that arose within our interaction with practitioners:

1. We observed that many of the activities that practitioners perform are based on implicit knowledge identified as experience or domain knowledge. This implicit knowledge is hard to grasp, describe, and often differs in specialization and complexity from the knowledge in the literature. Through our interaction with practitioners, we identified that the perception of what constitutes a dependency was part of the implicit knowledge. Also, in the literature, if definitions are provided, they usually vary widely. Thus we found the need to get an overview of the existing definitions of a 'dependency' and see how these definitions matched the implicit definitions of the practitioners.
2. Our interaction with practitioners started assuming that dependency analysis was useful. However, we soon realized that we needed explicit evidence to support

Table 1 Research questions and motivation

RQ1. What are the proposed definitions of dependencies?
Motivation: In order for dependency analysis solutions to meet practical requirements, the starting point is to understand what constitute a dependency.
RQ2. Why is dependency analysis needed (application areas)?
Motivation: Identify the needs/issues/problems that can be addressed using dependency analysis within the context of developing software-intensive systems.
RQ3. What are the available dependency analysis solutions?
Motivation: Obtain an overview of the existing solutions and the required resources in order to be able to complement, build on top or reuse these solutions.
RQ4. Are the proposed definitions and solutions usable in practice?
Motivation: Evaluate existing definitions and solutions based on the characteristics and requirements of software-intensive systems.

our assumption and convince the practitioners. We needed evidence to show why dependency analysis is necessary and useful for practitioners. Thus, we found the need to get an overview about the typical use cases and application areas for which researchers have developed dependency analysis solutions. More importantly, we wanted to find out how these areas matched the actual needs of practitioners.

3. Practitioners were using and testing several solutions to support the architecting process. They concluded that none of these solutions provided the desired support for dependency analysis. The goal of the practitioners was to find out how to solve particular dependency analysis problems using available resources. Practitioners often decide for solutions that use available and less expensive resources. Therefore, we had to present them with the problems that current solutions solve and what resources they require.

4. Finally, the goal of closely working together with practitioners was to identify which definitions and solutions can be useful and applicable according to their needs.

2.3.1 Target Audience

As one of the goals of the project is to support the SWAT at Philips Healthcare with identifying dependencies in their software system, we aimed at making the results of this review usable for them and, more in general, for people working with large, software-intensive systems. Next to that, the results of this review should provide dependency analysis researchers a better insight in the needs that practitioners have, the subjects that have already been covered, and the subjects that still require attention and could be useful in everyday practice.

3 Design of the Review

The protocol for conducting our review is based on the guidelines for systematic literature reviews as proposed in Kitchenham (2004a). Figure 3 illustrates the protocol which distinguishes four main phases: study search and selection, data extraction, data synthesis, and interpretation. The last phase, interpretation, is an addition to
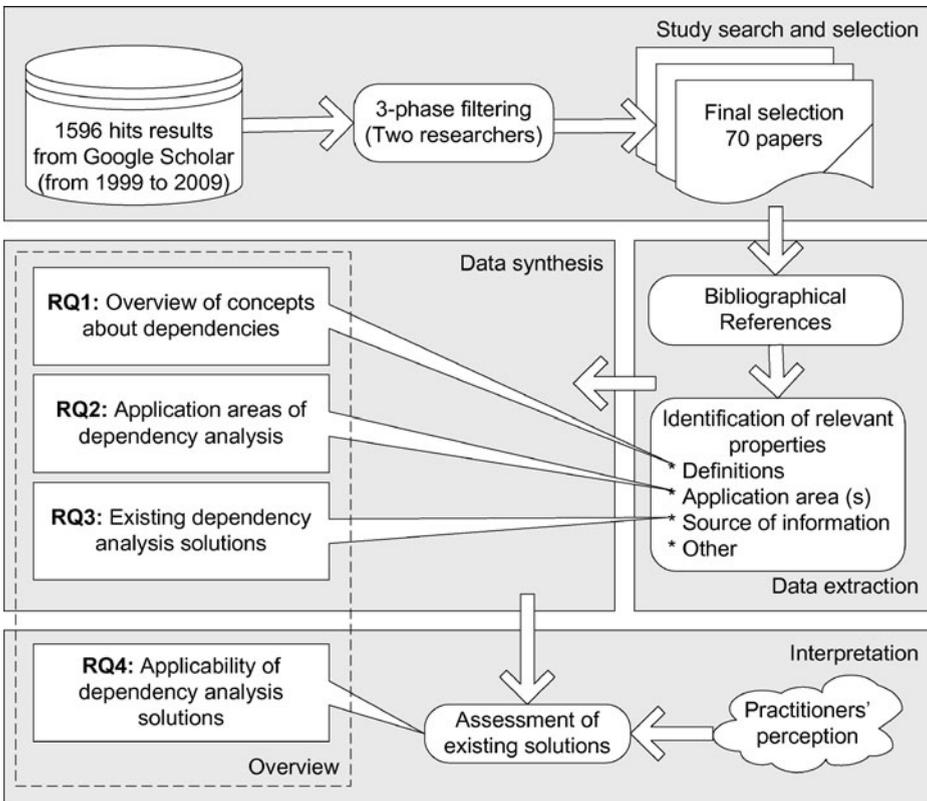
Fig. 3 Overview of the systematic review process

the proposed guidelines (Kitchenham 2004a), which was necessary to assess the applicability of the study results by practitioners. In the remainder of this section we describe the motivation and settings for the execution of each of the phases in the protocol. The threats to the validity of this study are discussed in Section 8.

3.1 Study Search and Selection

In this phase, we focused on the process to search and select articles from the literature. The process includes an automatic keyword search strategy and a filtering of the search results. This process enabled the selection of 70 articles.

3.1.1 Study Search

Our search process employs an automatic keyword search strategy using Google Scholar[1] as the search engine. We choose this combination because we want practitioners to be able to replicate our search and find the articles using accessible

---

resources. Further motivation and discussion of this choice is detailed in Section 8. To limit the amount of papers from areas of research other than computer science, we enabled an advanced search option in Google Scholar which tries to limit the subject area to papers from "engineering, computer science, and mathematics". In addition, we set the search option to search articles published over the last ten years. We experimented with several different search queries and, in the end, used the following three:

1. ```
"+(dependency OR dependence OR dependencies) analysis"
"software (system OR program)"
```
2. ```
"dynamic|static|behavioral|structural
dependence|dependency" "analysis|identification|"
+software "program|system"s
```
3. ```
+software intitle:Describing|Analyzing|Extracting
|Representing|Tracking|Using intitle:dependencies
|dependency
```

The first query is designed to find papers related to dependency analysis on software. Our preliminary investigation showed that the word dependency occurs in three forms together with the word analysis. Therefore, we require that at least one of the forms occurs in the search result. This query produced a set $SR_1$ of 703 search results.

The second query looks for papers mentioning specific types of dependencies. As we will explain in Section 4.2, several types of dependencies exist. Usually, the focus of a particular study is on a subset of these types. This query therefore tries to identify papers which mention at least one of the dependency types. This query produced a set $SR_2$ of 818 search results.

The third query is more restrictive than the first two. This query searches for software-related articles that explicitly state, i.e., in the title, that they are doing something with dependencies. The result of this query was a set $SR_3$ of 204 search results.

We designed the first two search queries in a initial phase of the study (pilot). The third search string was designed taking into account the results of the pilot phase. Figure 4 zooms into the study search and selection phase of the review protocol. In particular, we illustrate the processing of the search results through three filtering phases (pilot selection, final selection, and quality assessment). For each phase, we describe the input (sets of search results or selected papers from previous phase), the output (the number of candidate, in conflict, and excluded papers), number of reviewers, and the set of selected papers.

### 3.1.2 Study Selection

The study selection was implemented as an exclusion process of two phases, i.e. pilot selection and final selection, evaluating the three sets of search results (see Fig. 4). The pilot selection consists of two filters (1a and 1b). The input for this phase was the union of the first two sets of search results: 1,423 unique search results and 98 duplications.

In filter 1a, we excluded results that were obviously false positives. False positives include results from other fields than software engineering and computer sciences that Google Scholar did not filter. The first two authors reviewed the search results
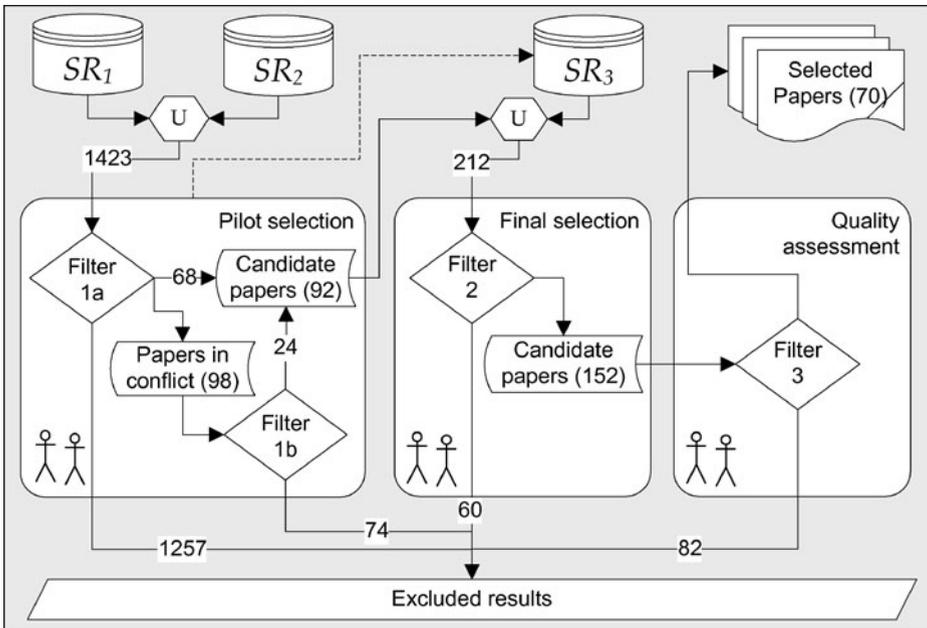
**Fig. 4** Study selection and quality assessment process

independently from each other looking at the title and the venue of the paper linked by the search result. The output of this step was the common selection of 68 candidate papers, the exclusion of 1,257 results, and 98 results with no common agreement (marked as Papers in conflict). This set of papers in conflict was the input for filter 1b where we conducted a shared discussion on each paper. The output of filter 1b increased the set of candidate papers to 92 and the number of excluded results to 1,331. After this filter, we concluded the pilot selection and designed the third query string taking into account the results from the shared discussion.

The final selection took place through the use of filter 2. The input for this filter was the union of the third set of search results and the set of candidate papers from the pilot selection: 212 unique search results and 84 duplications. We followed the same process as in the pilot selection scanning the titles, venues, and abstracts. We extended the false positive criteria excluding papers that have relation to the domain of computer science but not software engineering, e.g. bio-informatics (Fundel et al. 2007), and papers that use dependency analysis for different purposes than for analyzing software, e.g. state/event model checking (Lind-Nielsen et al. 2001). The output consisted of 152 candidate papers and the 60 false positives.

### 3.1.3 Quality Assessment

We conducted a third filter to assess the quality and relevance of the 152 candidate papers. *The quality criteria* that we used were based on three properties that a paper should have to fit in the context of our review. First, a paper provides a definition or description of the addressed dependency. Second, a paper provides information about the use cases or application area of the proposed solution. Third, a paper was

peer-reviewed and published at a venue related to the field of software engineering. The scores that we used for the first two properties are: Y (yes) when the definition or description is explicit, P (partly) when the definition or description is implicit, and N (no) when the definition or description can not be readily inferred.

Each of the first two authors independently annotated the papers with the description and score of the properties. Then, we compared, discussed, and resolved differences between individual annotations. This process enabled the identification of a set of four papers with the same authors and equal content but, with different titles, abstracts, and venues. The output of the quality assessment was the final selection of 70 papers and the exclusion of 82 papers, including the identified duplications (see Fig. 4). Table 2 lists the selected papers grouped by the venue and the respective venue type. In Tables 7, 8 and 9, the columns Definition and Application area match the selected papers to our quality assessment properties. According to our judgment, the selected 70 papers are those that provide the most clear definitions of dependency or explicitly state the purpose of the solution described in the paper.

### 3.2 Data Extraction

The data extraction was a manual process. We divided the selected papers into two sets and the first two authors processed one set each. We extracted two types of data from each of the selected papers (see Fig. 3). First, the bibliographical reference including the paper's title, authors, venue (journal or conference), and the URL for the digital version. Second, we extended the annotations made for the quality assessment (see Section 3.1.3) to identify a set of relevant properties in each paper regarding our research questions:

– For RQ1, a set of definitions about dependencies and types of dependencies.
– For RQ2, a list of use cases or development activities that dependency analysis contribute to.
– For RQ3, the types of sources of information used by dependency analysis solutions.

### 3.3 Data Synthesis

In this phase, we summarized and tabulated the extracted data following a bottom-up process. We aimed at producing the foundations for the overview (see the transition between data extraction and data synthesis in Fig. 3). Thus, we focused on the analysis of the extracted data to answers our first three research questions (see Table 1). The results are presented in Sections 4, 5, and 6 as summaries and categorizations that practitioners can use as overviews at first and then, if needed, as links or references to investigate details.

### 3.4 Interpretation

In the interpretation phase we aimed at collecting the information to answer our fourth research question (see Table 1). The interpretation process started with presenting the summaries for RQ1 (see Section 4), RQ2 (see Section 5), and RQ3 (see Section 6) to the practitioners. We iterated several times to agree on the content and format of the summaries, and to capture the practitioners' perception. We

**Table 2** Venues and selected papers

| Type | Acronym | Venue and selected papers |
|---|---|---|
| Conference | APAQS | Asia-Pacific Conference on Quality Software (Chen et al. 2000) |
| | ASE | IEEE/ACM International Conference on Automated Software Engineering (Breivold et al. 2008; Vieira and Richardson 2002) |
| | ASPLOS | International conference on Architectural support for programming languages and operating systems (Narayanasamy 2006) |
| | CAiSE | International Conference Advanced Information Systems Engineering (Khan et al. 2008) |
| | CASCON | Conference of the Center for Advanced Studies on Collaborative research (Ronen et al. 2006) |
| | COMPSAC | International Computer Software and Applications Conference (McComb et al. 2002; Moraes et al. 2005) |
| | CSMR | European Conference on Software Maintenance and Reengineering (Xiao and Tzerpos 2005) |
| | ICAC | International Conference on Autonomic Computing (Li et al. 2005b) |
| | ICCS | International Conference on Conceptual Structures (Cox et al. 2001) |
| | ICCSA | International Conference on Computational Science and Applications (Mao et al. 2007) |
| | ICFCA | International Conference Formal Concept Analysis (Pfaltz 2006) |
| | ICPC | International Conference on Program Comprehension (Lienhard et al. 2007) |
| | ICSE | International Conference on Software Engineering (Law and Rothermel 2003a; Maule et al. 2008; Vieira et al. 2001; Zimmermann and Nagappan 2008) |
| | ICSEA | International Conference on Software Engineering Advances (Alzamil 2007) |
| | ICSM | International Conference on Software Maintenance (Balmas et al. 2005; Binkley and Harman 2005; Cossette and Walker 2007; Dong and Godfrey 2007; Eisenbarth et al. 2001; Hassan and Holt 2004; Ishio et al. 2004; Jász et al. 2008; Korel et al. 2002) |
| | ISCC | International Conference on Computers and Communications (Keller et al. 2000) |
| | MODELS | International Conference on Model Driven Engineering Languages and Systems (Garousi et al. 2006) |
| | OOPSLA | Conference on Object-oriented programming, systems, languages, and applications (Sangal et al. 2005) |
| | OTM | On the Move to Meaningful Internet Systems Confederated Conferences (Xiao and Urban 2008) |
| | PDPTA | International Conference on Parallel and Distributed Processing Techniques and Applications (Keller and Kar 2000) |
| | QSIC | International Conference on Quality Software (Liangli et al. 2006) |
| | SERA | International Conference on Software Engineering Research, Management and Applications (Huang and Song 2007; Vasilache and Tanaka 2005) |
| | TCS | International Conference on Testing Computer Software (Ryser and Glinz 2000) |
| | VLDB | International conference on Very large data bases (Steinle et al. 2006) |
| | WCRE | Working Conference on Reverse Engineering (Callo Arias et al. 2008; Moise and Wong 2005) |

**Table 2** (continued)

| Type | Acronym | Venue and selected papers |
|---|---|---|
| Journal | SN | ACM Sigplan Notices (Chen et al. 2002; Li et al. 2005a) |
| | IJSEKE | International Journal of Software Engineering and Knowledge Engineering (Ivkovic and Kontogiannis 2006; Stafford and Wolf 2001; Xing and Stroulia 2006) |
| | IST | Information and Software Technology (Jiang et al. 2008) |
| | JSME | Journal of Software Maintenance and Evolution: Research and Practice (Glorie et al. 2009) |
| | NTCS | New Technologies on Computer Software (Zhao 2001) |
| | TACO | ACM Transactions on Architecture and Code Optimization (Tallam and Gupta 2007) |
| | TOSEM | ACM Transactions on Software Engineering and Methodology (Robillard 2008) |
| | TSE | IEEE Transactions on Software Engineering (Egyed 2003; Eisenbarth et al. 2003) |
| Symposium | SAC | ACM Symposium on Applied Computing1 (Bohnet et al. 2009) |
| | ESEM | International Symposium on Empirical Software Engineering and Measurement (Cataldo et al. 2008; Nagappan and Ball 2007) |
| | IM | IFIP/IEEE International Symposium on Integrated Network Management (Brown et al. 2001) |
| | ISCIS | International Symposium on Computer and Information Sciences (Jourdan et al. 2006) |
| | ISSRE | International Symposium on Software Reliability Engineering (Law and Rothermel 2003b; Zimmermann and Nagappan 2007) |
| | ISSTA | International symposium on Software testing and analysis (Xin and Zhang 2007) |
| | METRICS | International Symposium on Software Metrics (Leitch and Stroulia 2003) |
| | NOMS | Network Operations and Management Symposium (Gao et al. 2004) |
| | SFM | International School on Formal Methods: Software Architectures (Stafford et al. 2003) |
| Workshop | DSOM | IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (Agarwal et al. 2004; Gupta et al. 2003) |
| | IWPC | International Workshop on Program Comprehension (Chen and Rajlich 2000) |
| | IWPSE | International Workshop on Principles of Software Evolution (Zhao 2002) |
| | MSR | International Workshop on Mining Software Repositories (Kagdi and Maletic 2007) |
| | PASTE | ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (Zhang and Ryder 2007) |
| | VISSOFT | International Workshop on Visualizing Software for Understanding and Analysis (Holmes and Walker 2007) |

captured this perception by observing and asking practitioners about their concerns regarding the applicability and potential usage of the information presented in the summaries. Then, we used the collected perception to build the summary for RQ4 (see Section 7).

## 4 Overview of Concepts about Dependencies

Dependency analysis aims to make information about dependencies explicit and accessible, which is of paramount importance when changing or evolving a software system (Loyall and Mathisen 1993; Podgurski and Clarke 1990). However, this requires knowledge on what a dependency is. Therefore, we have looked at existing definitions of dependencies in the literature.

### 4.1 Definition of Dependencies in the Literature

Much of the present literature takes the definition of dependency for granted and where definitions are given, they vary widely. One of the first definitions of dependency in the literature of computer science was stated by Stevens et al. (1974): *a dependency is the degree to which each component relies on each one of the other components in the software system. The fewer and simpler the connections between components, the easier it is to understand each component without reference to other components.* This definition, introduced in 1974, has since been used by many authors and applied to various different areas.

Another definition, similar to the one provided by Stevens et al., is proposed by Vieira and Richardson (2002). They state that dependencies reflect the potential for one component to affect (via the various in and outputs) or be affected by the elements (e.g., other components, the platform on which it runs) that compose the system. Although this definition is similar to, it is not the same as the definition provided by Stevens et al. as Vieira and Richardson have removed the notion of strength from their definition.

Most of the more recent definitions (Cox et al. 2001; Loyall and Mathisen 1993; Mehta et al. 2000; Podgurski and Clarke 1990; Stafford and Wolf 1998) describe dependencies in software systems as relations between components. These dependencies, regardless of their complexity, provide mechanisms for transferring data, control, or both from one component to another. Transfer of control and data are often related to structures in the system source code like function calls and conditional statements. Unfortunately, these definitions are hard to use when looking at a system in other ways than by examining the source code. For instance, looking at the dynamic behavior of the system often involves abstractions which are not available in the source code.

Therefore, some authors choose more high-level definitions and look at dependencies as interactions between different managed objects or components which are only observable from outside of the application (Allen and Garlan 1997; Keller et al. 2000). These dependencies can even run between elements which are not part of the same system. This is especially the case when trying to identify so-called *operational dependencies* as described by Brown et al. (2001). An example of these dependencies

are dependencies between a web application, the web naming service it relies on, the underlying database, and the operating system.

### 4.2 Types of Dependencies

Besides the concept of dependencies, literature also describes various types of dependencies. However, we consider that they all fit into three main categories which we will discuss below.

*Structural Dependencies*   Often, when talking about dependencies, what is actually meant are structural (Allen and Garlan 1997; Stafford and Wolf 1998) dependencies among parts of a system. Structural dependencies have been widely discussed in the literature. Structural dependencies can be divided into several subcategories: content dependencies, common dependencies, external dependencies, control dependencies, stamp dependencies and data dependencies (also called data flow dependencies) (Allen and Garlan 1997; Balmas et al. 2005; Myers 1975; Podgurski and Clarke 1990; Stafford and Wolf 1998; Stevens et al. 1974). Although most structural dependencies can be found by inspecting the source code (i.e. static analysis of the source code), structural dependencies also exist on the level of models and application execution. An example of a structural dependency at the execution level is a web server which executes a diagnostic routine in order to determine whether there is a problem with the TCP/IP-stack that is provided.

*Behavioral Dependencies*   In contrast to structural dependencies, behavioral (Stafford and Wolf 1998) or interaction (Allen and Garlan 1997) dependencies often involve abstractions not directly provided by programming languages: use of public interfaces (e.g. external programs or devices), event broadcast, client-server protocols, temporal ordering, etc. (Allen and Garlan 1997; Li et al. 2005a; Mehta et al. 2000; Stafford and Wolf 1998). Using the previous example for dependencies at the application management level again, a behavioral dependency exists between the web server and the TCP/IP-stack as well, because the web server needs the TCP/IP-stack in order to perform its tasks.

*Traceability Dependencies*   In an iterative process, a developer cannot discard the requirements after the design is built nor can a developer discard the design after the source code is programmed (Egyed 2003). Therefore, developers have the need to maintain the inter-relationships between the different artifacts. These inter-relationships are called traceability dependencies and they characterize the dependencies between requirements, design, and code (Egyed 2003; Gotel and Finkelstein 1994; Watkins and Neal 1994). Traceability dependencies are different from the other two types of dependencies in that they do not represent dependencies between the same type of elements, i.e. between code elements or dynamic aspects of a program, but between different kinds of development artifacts.

## 5 Application Areas of Dependency Analysis

In this section we aim to answer our second research question, *Why is dependency analysis needed?* (see Table 1). Our answer to this question is an overview of a set

of activities that dependency analysis solutions in the literature claim to support or address. Our motivation to present this answer under these terms is motivated by what we observed in practice. We noticed that practitioners often relate their needs to the activities they perform within the different phases of a given development project. However, we also observed that these needs are often not explicit nor are they easy to identify. Especially from the research perspective, what specific activities practitioners follow and need support for, is an interesting topic.

We consider that providing this answer, even before identifying the various existing solutions, is useful for two reasons. First is to establish the communication with practitioners. And second is to identify whether dependency analysis support the activities that practitioners actually perform.

Table 3 lists various activities, identified through the review, that dependency analysis solutions in general claim to support. We have classified these various activities as a set of application areas that match to actual tasks conducted by practitioners within the development and maintenance of software systems. In addition, Table 3 provides references to the solutions which, according to our criteria, explicitly focus on the given application area. In the rest of this section we describe each of the identified application areas and how dependency analysis support them.

## 5.1 Application Level Analysis and Management

When systems are deployed in the field and used by end-users, system's components or applications result in more places and paths than those considered within the design and development. Application level analysis and management focuses on the behavior of systems in the field and related "end-user noticeable" system aspects, e.g., performance, availability, and other end-user-visible metrics (Brown et al. 2001). Information about dependencies in this context is often necessary and useful to support system managers and administrators who concern about the effects and propagation of system applications problems in the field. The available dependency analysis solutions to support this area (see Table 3) aim at identifying structural and behavioral dependencies between major elements of a running system (subsystems. applications, services, data repositories etc.). These solutions support practitioners in the management of end-user-reported problems conducting activities such as automated distributed management (Keller and Kar 2000), problem determination (Agarwal et al. 2004; Brown et al. 2001; Gao et al. 2004; Gupta et al. 2003; Li et al. 2005b), root cause analysis and fault localization (Steinle et al. 2006), and maintaining the correctness of concurrency in multi processes systems (Xiao and Urban 2008).

## 5.2 Architecture Description and Analysis

Due to the increasing size and complexity of software systems, architectural descriptions has become important assets for development organizations. Practitioners such as software architects and designers often construct and use architectural descriptions to facilitate the communication within the various stakeholders. Practitioners use architectural descriptions to extract information about many aspects of a system's structure and behavior, and including dependencies at an architecture

**Table 3** Overview of the application areas including relevant literature

| Application area | Includes | References |
|---|---|---|
| Application level analysis and management | Problem determination<br>Automated distributed management<br>Platform and components porting<br>Service management<br>Root cause and fault localization<br>Maintaining correctness of concurrent processes | (Brown et al. 2001; Li et al. 2005b; Gao et al. 2004;<br>Keller and Kar 2000; Ronen et al. 2006;<br>McComb et al. 2002; Gupta et al. 2003;<br>Keller et al. 2000; Steinle et al. 2006;<br>Agarwal et al. 2004; Xiao and Urban 2008) |
| Architecture description/ analysis | Architecture understanding<br>Code architecture analysis<br>Description of component-base systems<br>Analysis of formal architectural descriptions | (Vieira and Richardson 2002; Stafford and Wolf 2001;<br>Vieira et al. 2001; Stafford et al. 2003;<br>Zhao 2001; Sangal et al. 2005) |
| Change impact analysis | Change impact in aspect-oriented software<br>Change impact of component-based systems<br>Change prediction<br>Dependence clusters and dependence pollution<br>Dynamic impact analysis of object-oriented programs<br>Efficient source code navigation<br>Evolvability and modularity relation<br>Identification of class change profiles<br>Impact of database schema change<br>Independent development<br>Models and source code synchronization<br>Requirement change impact on architectural elements | (Zhao 2002; Mao et al. 2007; Hassan and Holt 2004;<br>Law and Rothermel 2003b; Kagdi and Maletic 2007;<br>Binkley and Harman 2005; Huang and Song 2007;<br>Robillard 2008; Breivold et al. 2008;<br>Xing and Stroulia 2006; Maule et al. 2008;<br>Glorie et al. 2009; Khan et al. 2008) |

| Category | Topics | References |
|---|---|---|
| Program/system understanding | Analysis of concurrency<br>Analysis of multi-languages/poly-lingual systems<br>Identification of dynamic control dependencies<br>Program slicing analysis<br>Reuse analysis | (Chen and Rajlich 2000; Chen et al. 2002; Cossette and Walker 2007; Moise and Wong 2005; Xin and Zhang 2007; Jiang et al. 2008; Holmes and Walker 2007; Jász et al. 2008; Pfaltz 2006) |
| Quality assurance, testing and debugging | Prediction of defects and failures<br>Explain post-release failures<br>Forecasting the load level of system components<br>Interclass testing<br>Model-based regression testing<br>Scenario-based testing<br>Test suite reduction<br>Aspect-oriented debugging<br>Bug introduction in C/C++ programs<br>Replay debugging for multi-threaded programs | (Garousi et al. 2006; Zimmermann and Nagappan 2007, 2008; Zhang and Ryder 2007; Korel et al. 2002; Ryser and Glinz 2000; Moraes 2005; Jourdan et al. 2006; Ishio et al. 2004; Bohnet et al. 2009; Narayanasamy 2006; Nagappan and Ball 2007) |
| Refactoring and modularization | Assessing modularization and productivity<br>Return on investment of refactoring<br>Undesired dependency removal<br>Use of design patterns and potential problems | (Leitch and Stroulia 2003; Alzamil 2007; Cataldo et al. 2008; Dong and Godfrey 2007) |
| Traceability and feature analysis | Identification of trace relationships<br>Link requirements analysis and design<br>Models and source code synchronization<br>Feature location and analysis | (Egyed 2003; Vasilache and Tanaka 2005; Ivkovic and Kontogiannis 2006; Chen et al. 2000; Eisenbarth et al. 2003; Eisenbarth et al. 2001; Lienhard et al. 2007) |

level. In the literature, a number of dependency analysis solutions are presented to support dependency analysis as part of activities related to the construction and use of architectural descriptions (see Table 3). These activities include, the analysis and understanding of formal architectural descriptions  (Stafford and Wolf 2001; Stafford et al. 2003; Zhao 2001), code architecture analysis (Sangal et al. 2005), and the description of large component-base systems (Vieira et al. 2001; Vieira and Richardson 2002).

## 5.3 Change Impact Analysis

An area in which dependency analysis is often applied is for assessing the impact of changes, i.e. change impact analysis (often just referred to as software mainte-nance). Change impact analysis is used by practitioners (e.g., architects, designers, developers, and testers) who often need to assess the effect of a change in the system they develop or maintain. This analysis is important because changes in one part of a system do not stand on their own, but require further modifications in other parts of the system.

Dependency analysis support change impact analysis in many different ways (see Table 3). First, some solutions are presented to support change impact analysis on specific type of systems, e.g., aspect-oriented (Zhao 2002), component-based (Mao et al. 2007), and object oriented (Huang and Song 2007; Xing and Stroulia 2006). Second, many solutions are for supporting activities that facilitate change impact analysis, e.g., change prediction (Hassan and Holt 2004; Kagdi and Maletic 2007; Law and Rothermel 2003b), identification of dependence clusters and dependence pollution (Binkley and Harman 2005), dynamic impact analysis in object-oriented programs (Huang and Song 2007), identification of class change profiles (Xing and Stroulia 2006), impact of database schema change (Maule et al. 2008), and efficient source code navigation (Robillard 2008). Third, a number of other solutions are presented to support change impact on development aspects such as the relation between evolvability and modularity (Breivold et al. 2008), independent develop-ment (Glorie et al. 2009), and requirement change impact on architectural elements (Khan et al. 2008).

## 5.4 Program/System Understanding

It is well-known that within development and maintenance, practitioners spend a considerable amount of time studying artifacts such as source code and documen-tation. This is often necessary to gain a sufficient level of understanding about the system they develop or maintain. Program and system understanding, therefore, is another popular area that dependency analysis solutions support (see Table 3). Dependency analysis is presented to support the understanding of systems like con-current software systems (Chen and Rajlich 2000) and multi-languages/poly-lingual systems (Cossette and Walker 2007; Moise and Wong 2005). The main activities that enable program understanding and are supported by dependency analysis include the identification of particular types of dynamic dependencies (Jász et al. 2008; Pfaltz 2006; Xin and Zhang 2007), program slicing analysis (Jiang et al. 2008), and reuse analysis (Holmes and Walker 2007).

## 5.5 Quality Assurance, Testing and Debugging

Quality attributes include reliability, availability, safety, and performance. Assuring these quality attributes in a software system is an important concern for practitioners, especially when developing dependable software-intensive systems (e.g., command and control systems, aircraft aviation systems, robotics, and nuclear power plant systems). Testing and debugging are the usual activities that development organizations perform to verify and assure the quality of such systems. Dependency analysis support support quality assurance through various planning and implementation activities (see Table 3).

In the planning, dependency analysis support the forecasting of load levels of system components (Garousi et al. 2006), and prediction of defects and failures (Zimmermann and Nagappan 2007, 2008). Forecasting the load level of system components is supported by analyzing behavioral dependencies on model designs and aims at devising appropriate provisions for the most dependable entities of a system before implementation. Also, practitioners (e.g. managers) could identify in advance the system or program units that are more likely to face defects and compromise the system's quality. With this information in advance, practitioners can estimate the time and cost for the design and execution of testing activities, especially for the units that may need to be tested the most. Some of the main testing activities supported by dependency analysis are design of interclass testing (Zhang and Ryder 2007), model-based regression testing (Korel et al. 2002), scenario-based testing (Ryser and Glinz 2000), and test suite reduction (Jourdan et al. 2006).

Dependency analysis solutions can also be used to explain post-release failures (Nagappan and Ball 2007) and to deliberately insert faults into source code (Moraes et al. 2005) to accelerate errors and failures situations. These activities are useful to observe the system's behavior in the presence of faults and therefore identify the source code elements that should be monitored for debugging during the implementation. Debugging is looking for the anomalies in the code, which are syntactic patterns that evidence a programming error or irrespective use of the language specification, e.g., using a variable before it has been defined. Dependency analysis supports debugging for looking for various kinds of anomalies in program statements (Podgurski and Clarke 1990), bug introduction in C/C++ programs (Bohnet et al. 2009), replay debugging for multi-threaded programs (Narayanasamy 2006), and even debugging of aspect-oriented software (Ishio et al. 2004).

## 5.6 Refactoring and Modularization

A long-standing technique for improving an existing design is diligent restructuring through local code transformations, commonly knows as "Refactoring" (Fowler 1999). In the area of refactoring, dependency analysis solutions provide support for predicting the Return on Investment (ROI) for possible design restructuring (Leitch and Stroulia 2003). It is often hard for practitioners decide and quantify the trade-off between the up-front cost of restructuring and the expected downstream savings. Similar to refactoring, taking decisions about modularization is hard in practice.

Modularization is often considered as a beneficial technique to reduce inter-dependencies among the components of a system. Dependency analysis supports practitioners in the identification of dependencies that should be taken into account

for modularization and reduction of work dependencies (Cataldo et al. 2008). The support includes how to detect and model the kind of dependencies that go against modularization and are candidates to be removed. The first kind are the relationships that appear due to the particularities of the implementation paradigm and design (Dong and Godfrey 2007). The second kind are redundant relationships that create undesired coupling between implementation modules, which often do not contribute to the respective system function output (Alzamil 2007).

## 5.7 Traceability and Feature Analysis

Software development artifacts such as model descriptions, specifications, and source code are highly interrelated. Changes in one artifact effect another thus setting in motion a cascade of changes. Trace dependencies characterize such relationships in an abstract fashion. A common problem in practice is that the absence of information about trace dependencies or the uncertainty of its correctness, limits the usefulness of software models during software development activities. In the literature several solutions are presented to address this situation (see Table 3). The support include automated approaches to generate and validate trace dependencies (Egyed 2003), solutions to link the result of requirement analysis, i.e. scenario descriptions, with model designs (Vasilache and Tanaka 2005), and solutions to synchronize design models and the respective implementation code (Ivkovic and Kontogiannis 2006).

Similar to the identification of trace information, it is important to identify which parts of the source code implement a given system feature (system's externally visible behavior). This information is in general not obvious or outdated, which causes that understanding the system becomes harder every time a feature is changed. Thus dependency analysis solutions provide support for the identification and analysis of system features mainly by conducting scenario-based analysis (Chen et al. 2000; Eisenbarth et al. 2001, 2003; Lienhard et al. 2007).

## 6 Existing Dependency Analysis Solutions

In this section, we present the answer to our third research question (see Table 1) describing existing dependency analysis solutions. To structure the description, we classify the existing solutions by their source of information. Dependency analysis solutions take the source of information as input data and transform it into information at a higher level of abstraction. Information at a higher level of abstraction is then be used to reason about the dependencies and to solve issues in the various application areas (see Section 5). The sources of information used among existing dependency solutions can be classified in three groups: source code, descriptions and models, and run-time monitored and configuration data.

There are alternative criteria to classify dependency analysis solutions, e.g., the kind of information output, the required interaction, or the degree of user intervention. However, our decision to classify dependency analysis solutions by the source of information was driven by the resource constrained perspective of practitioners (see Section 2.3). Presenting existing solution by their source of information helped us to make the required resources explicit, i.e. the sort of data that practitioners may require or should make it available to use a given solution.

6.1 Source Code-based Solutions

Source code is, if not the most popular, the most well-known source used by dependency analysis solutions. Source code provides syntactic and semantic information that describes the implementation and the structure of a software system. Syntactic and semantic information in the source code are respectively represented by the abstract syntax tree and abstract semantic graphs. Both, semantic and syntactic code information describe code artifacts (e.g. variables, operators, methods, classes) and relationships between them.

Dependency analysis solutions using code information are often used to identify structural dependencies at different levels of abstractions (e.g. program statements, module, and file level). Among solutions that analyze source code data one can distinguish three groups based on the analysis approach: static, dynamic, and change history analysis (see Table 4).

In Table 4, we illustrate the match between the analysis approach (or combination of approaches), the application area, and the reference to the identified dependency analysis solution(s). Although the identified approaches are very different, these approaches share two main underlying characteristics. First, the identification and description of dependencies is bases on the Program Dependency Graph (PDG) (Ferrante et al. 1987; Podgurski and Clarke 1990). PDG is a classic dependency model to identify data and control dependencies between program statement elements (variables, operators, and operands). Second, the identified relationships or dependencies link source code related artifacts, but at different levels of abstraction (e.g. program statements, classes, modules, and even groups of source code files).

### 6.1.1 Static Analysis

Dependency analysis solutions based on static analysis are extensions of the PDG (Chen and Rajlich 2000; Chen et al. 2000; Ishio et al. 2004; Leitch and Stroulia 2003; Zhao 2002) and the Dependency (or Design) Structure Matrix (DSM) (Breivold et al. 2008; Sangal et al. 2005). A variety of static analysis techniques are used by dependency solutions to discover and organize dependency information in graphs and matrices. These techniques are approximation algorithms (Zhang and Ryder 2007), context-sensitivity dataflow (Maule et al. 2008), formal concept analysis and clustering (Glorie et al. 2009), island grammars (Cossette and Walker 2007), search-based slicing (Jiang et al. 2008), source code navigators for heterogeneous code (Moise and Wong 2005), topology analysis (Robillard 2008), and annotations and navigation models (Holmes and Walker 2007). The goal of these techniques is to reflect the system structure and highlight patterns and problematic relationships that practitioners deal with through various application areas (see Table 4).

### 6.1.2 Dynamic Analysis

Dependency analysis solutions based on dynamic analysis use source code-based data in the form of execution traces. An execution trace can be identified as function, procedure, or method being called. Execution traces are collected using techniques such as source code instrumentation, platform profiling, and compiler profiling. Most techniques and tools for execution trace analysis are presented for specific paradigms and even specific programming languages (Hamou-Lhadj and Lethbridge 2004).

**Table 4** Dependency analysis solutions using source code-based information

| Approach | Application areas | References |
| --- | --- | --- |
| Static analysis | Architecture description/analysis | (Sangal et al. 2005) |
| | Change impact analysis | (Binkley and Harman 2005; Breivold et al. 2008; Glorie et al. 2009; Hassan and Holt 2004; Maule et al. 2008; Robillard 2008; Zhao 2002) |
| | Program/system understanding | (Chen and Rajlich 2000; Cossette and Walker 2007; Holmes and Walker 2007; Jiang et al. 2008; Moise and Wong 2005) |
| | Feature analysis | (Chen et al. 2000; Glorie et al. 2009) |
| | Quality assurance and testing | (Ishio et al. 2004; Zhang and Ryder 2007) |
| | Refactoring | (Dong and Godfrey 2007; Leitch and Stroulia 2003) |
| Dynamic analysis | Change impact analysis | (Huang and Song 2007; Law and Rothermel 2003a, b; Tallam and Gupta 2007) |
| | Program/system understanding | (Jász et al. 2008; Pfaltz 2006; Xin and Zhang 2007) |
| | Refactoring and modularization | (Alzamil 2007) |
| | Traceability and feature analysis | (Egyed 2003; Lienhard et al. 2007) |
| Change history analysis | Change impact analysis | (Kagdi and Maletic 2007) |
| | Refactoring and modularization | (Cataldo et al. 2008) |
| | Debugging | (Nagappan and Ball 2007) |
| Combined static and dynamic analysis | Application level analysis | (Ronen et al. 2006) |
| | Quality assurance | (Zimmermann and Nagappan 2007, 2008) |
| | Feature analysis | (Eisenbarth et al. 2001, 2003) |
| Combined static, dynamic, and change history analysis | Refactoring and modularization, change impact analysis, and debugging | (Bohnet et al. 2009) |

In the cases of dependency analysis, most solutions are to analyze object oriented implementations exploring relationships such as inheritance, polymorphism, and dynamic binding of languages such as Java and C++ (Egyed 2003; Lienhard et al. 2007).

The dynamic analysis techniques used among dependency analysis solutions include, footprint graph analysis (Egyed 2003), clustering (Xiao and Tzerpos 2005), whole path profiling (Law and Rothermel 2003a, b), object flow analysis (Lienhard et al. 2007), redundant coupling detection (Alzamil 2007), online detection (Xin and Zhang 2007), execute after/before analysis (Jász et al. 2008), compression and traversing of traces (Tallam and Gupta 2007), and formal concept analysis (Pfaltz 2006). These techniques enable dependency analysis in two ways. First, these techniques

identify relationships between object oriented code artifacts, e.g., objects, classes, and methods that happen at run-time. Second, these solutions also identify traceability dependencies between system features, execution scenarios, design models, and object oriented code artifacts. Table 4 illustrates the application areas supported by these various techniques. For a more exhaustive analysis of solutions that analyze execution traces, without a particular focus on dependencies, we refer the reader to Hamou-Lhadj and Lethbridge (2004).

### 6.1.3 Historical Analysis

Dependency analysis solutions based on historical analysis use the change history of source code artifacts. Change history of source code artifacts provides information about change patterns, e.g. a set of code files that were changed together frequently in the past. Change patterns are the relationships that solutions in this group characterize as dependencies. The identified historical analysis techniques include the analysis of modification requests (Cataldo et al. 2008), co-change mining (Kagdi and Maletic 2007), and analysis of churn metrics (Nagappan and Ball 2007). Table 4 illustrates the application areas supported by these various techniques.

### 6.1.4 Combining Information Sources

Some dependency analysis solutions propose combinations of analysis approaches to increase the completeness and precision of the identified information about dependencies. We identified two main combinations: static with dynamic analysis, and static with dynamic and change history, which are applied to support various application areas (see Table 4).

The techniques used by solutions that combine static and dynamic analysis include concept analysis (Eisenbarth et al. 2001, 2003), pattern languages (Ronen et al. 2006), and network analysis (Zimmermann and Nagappan 2007, 2008). Concept analysis is applied using a scenario-based approach that combines execution traces and static relationships. A pattern language is applied to abstract execution traces into relationships that describe the access to system level resources, e.g. databases, message queues, and control systems. Network analysis is applied to track dependency information at the function level (including calls, imports, exports, RPC, COM, and Registry accesses) and present it at the level of binaries and system modules.

A solution that combines static with dynamic and change history analysis is presented in Bohnet et al. (2009). The technique of this solution focuses on the reduction of change sets (historical information) by projecting them onto execution traces and static relationships that involve the source code artifacts in the change sets.

## 6.2 Descriptions and Model-based Solutions

System documentation often includes diagrammatic and semi-formal descriptions that are created to describe the structure and behavior of a software system at a high level of abstraction. Diagrammatic descriptions include representations such as chart UML diagrams and sketches with blocks and arrows. Semi-formal descriptions include descriptions using Architectural Description Languages (ADLs) and Interface Description Languages (IDLs), which are initiatives of the software architecture

research community and component-based platform respectively to describe systems at an architectural level.

Table 5 lists the identified dependency analysis solutions that use diagrammatic representations and semi-formal descriptions to identify dependencies (behavioral and structural) at a high level of abstraction such as architectural level, rather than at the level of source code artifacts.

### 6.2.1 Diagrammatic Descriptions

Dependency analysis solution work with various types of diagrammatic representations such as top-down descriptions (McComb et al. 2002), component-based models (Vieira and Richardson 2002), matrix models (Khan et al. 2008; Mao et al. 2007; Xing and Stroulia 2006), chart diagrams (Garousi et al. 2006; Moraes et al. 2005; Ryser and Glinz 2000), and business process models (Ivkovic and Kontogiannis 2006; Vasilache and Tanaka 2005). Top-down descriptions are used to construct dependency models for application level analysis (McComb et al. 2002). Component-based models are used to construct Component Based Dependency Models (CBDM) for describing and inferring data/control and direct/indirect dependencies between components (Vieira and Richardson 2002), and between components' access points, i.e. interfaces and ports but using semi-formal interface specifications (Vieira et al. 2001). Matrix models are the basis for the construction of dependency matrices to analyze changes in UML models and detect design-level structural modification (Khan et al. 2008; Mao et al. 2007; Xing and Stroulia 2006).

Dependency analysis solutions use chart diagrams for several purposes. Model-Based Behavioral Dependency Analysis (MBBDA) (Garousi et al. 2006) derive behavioral dependency information from UML design models. Chart diagrams are also used to identify chaining in class interfaces from class diagrams and support quality assurance and testing activities (Moraes et al. 2005; Ryser and Glinz 2000). A final

**Table 5** Dependency analysis solutions using description and model-based information

| Approach | Application areas | References |
|---|---|---|
| Analysis of diagrammatic descriptions | Application level analysis and management | (McComb et al. 2002) |
| | Architecture description/analysis | (Vieira and Richardson 2002) |
| | Change impact analysis | (Mao et al. 2007; Khan et al. 2008; Xing and Stroulia 2006) |
| | Quality assurance | (Garousi et al. 2006; Ryser and Glinz 2000; Moraes 2005) |
| | Traceability | (Ivkovic and Kontogiannis 2006; Vasilache and Tanaka 2005) |
| Analysis of semi-formal descriptions | Architecture description/analysis | (Stafford and Wolf 2001; Zhao 2001; Vieira et al. 2001; Stafford et al. 2003) |
| | Testing | (Korel et al. 2002; Jourdan et al. 2006; Liangli et al. 2006) |

set of solutions using diagrammatic representations apply formal concept analysis to cluster objects in business process models that can be considered dependent (Ivkovic and Kontogiannis 2006; Vasilache and Tanaka 2005). The cluster identified by these solutions enable the analysis of the traceability between requirement analysis and design.

### 6.2.2 Semi-formal Descriptions

Solutions that use semi-formal descriptions written on ADLs include chaining analysis (Stafford and Wolf 2001; Stafford et al. 2003) and the construction of architectural dependency graphs (ADG) (Zhao 2001). Chaining analysis is used to identify behavioral dependencies using syntactic and structural information of descriptions in an ADL. The ADG solution uses the syntactic and structural information of descriptions written in the Acme ADL for identifying component-component dependencies. Chaining and ADG analyze ADL's descriptions in a similar way than the solutions that analyze source code, i.e., analysis of semantic and syntactic information. However, in contrast to source code, ADL's source elements represent architecture level elements of the software system such as components, connectors, and ports.

Extended Finite State Machine (EFSM) models (Jourdan et al. 2006; Korel et al. 2002) are another semi-formal descriptions that describe software systems and are used by some dependency analysis solutions. Dependency analysis enables the analysis of differences between a original ESFM model and a modified model, which help to identify the modified elements and support the reduction of regression testing activities.

### 6.3 Run-time Monitored and Configuration-based Solutions

When analyzing an existing system either functioning in the field or under testing, different sources of information appear available. These sources include run-time monitored data and configuration repositories, which provide information about the execution and setting of the system components.

Table 6 lists a set of dependency analysis solutions that use run-time monitored data and configuration repositories. The solutions in this set share a number of common characteristic. First, most of these solutions are dedicated to support application level analysis and management activities (see Section 5.1). Second, these solutions see major elements of a running system (subsystems, applications, services, data

**Table 6** Dependency analysis solutions using configuration and actual execution-based information

| Approach | Application areas | References |
| --- | --- | --- |
| Analysis of monitored data | Application level analysis and management | (Agarwal et al. 2004; Li et al. 2005b; Steinle et al.2006; Xiao and Urban 2008; Brown et al. 2001; Gupta et al. 2003; Gao et al. 2004; Callo Arias et al. 2008) |
|  | Debugging | (Narayanasamy 2006) |
| Analysis of configuration repositories | Application level analysis and management | (Keller et al. 2000; Keller and Kar 2000) |

repositories etc.) as black boxes. Third, these solutions focus on the identification and analysis of structural and behavioral dependencies between major system elements.

### 6.3.1 Run-time Monitored Solutions

Run-time monitored data describe events that happen within the execution of a system (e.g. errors, warnings, resource usage). This data is captured by the system infrastructure or with the facilities of the run-time platform (e.g. operating system, middleware, virtual machine). Logging is the most popular system infrastructure among software systems to collect monitored data. Logging data is stored in repositories with system specific formats, which dependency analysis solutions analyze using data mining algorithms to support offline construction of dependency models (Callo Arias et al. 2008; Steinle et al. 2006; Xiao and Urban 2008). Self healing systems provide monitoring infrastructure similar to logging that dependency analysis solutions use to support online construction and analysis of dependency matrices (Gao et al. 2004).

Most run-time platforms provide built-in instrumentation for monitoring statistic data about system and platform resources, e.g., invocation and average execution time counters, which are primarily used for accounting and performance tuning purposes. The monitored statistics are stored in system repositories with generic formats for all systems running on a given platform. Dependency analysis solutions first populate these repositories using fault injection and perturbation of system components. Then, the solutions analyze the repositories using data-mining algorithm, statistical analysis, (Agarwal et al. 2004; Brown et al. 2001; Gupta et al. 2003; Li et al. 2005b). Other solutions use hardware-based monitoring mechanisms to capture shared memory dependencies for supporting debugging activities (Narayanasamy 2006).

### 6.3.2 Configuration Repositories

Configuration repositories provides information about the setting and configuration of the elements and environment of software systems. For example, system configuration repositories that keep track of the installed software packages, filesets, and their versions are the Windows Registry of Microsoft Windows platforms, AIX Object Data Manager (ODM) on IBM AIX platforms, and DPKG on Linux/Debian platforms. Configuration repositories can be analyzed to extract functional and structural information for dependency analysis (Keller et al. 2000; Keller and Kar 2000). For instance, functional information helps to identify the system available services (e.g. database service, name service, end-user application service etc.) and structural information the technical descriptions of the characteristics of software components that realize the identified services.

## 7 Applicability of Dependency Analysis

In this section we present the summary of the information that we captured in the interpretation phase (see Section 3.4), with respect to the applicability of definitions related to dependencies and the types of dependency analysis solutions in practice.

The former is presented in Section 7.1. The latter is elaborated in Sections 7.2, 7.3 and 7.4, where each section corresponds to one type of dependency analysis solutions and is divided in two parts:

– First, we summarize the value, practitioners' concerns, and opportunities for improvement that we identified for the given type of dependency analysis solution.
– Second, we complement the quality assessment of the selected papers (Section 3.1.3) with our assessment about each paper's applicability in practice (see columns 'Case', 'Easy to Use', and 'Ready to Use' in Tables 7, 8 and 9). This quality assessment is in turn based on two factors: the literature and the

**Table 7** Assessment of studies using source code-based information

| Reference | Definition | Application area | Case | Easy to use | Ready to use |
|---|---|---|---|---|---|
| Alzamil (2007) | P | Y | Toy | P | N |
| Binkley and Harman (2005) | Y | P | OS | P | N |
| Bohnet et al. (2009) | Y | Y | Industrial | Y | N |
| Breivold et al. (2008) | Y | Y | Industrial | P | P |
| Cataldo et al. (2008) | P | P | None | N | N |
| Chen and Rajlich (2000) | P | P | OS | P | P |
| Chen et al. (2000) | Y | P | None | N | N |
| Cossette and Walker (2007) | P | Y | OS | P | N |
| Dong and Godfrey (2007) | Y | P | OS | N | N |
| Egyed (2003) | Y | Y | Industrial | N | P |
| Eisenbarth et al. (2001) | P | Y | OS | P | P |
| Eisenbarth et al. (2003) | Y | Y | OS | N | N |
| Glorie et al. (2009) | Y | Y | Industrial | N | P |
| Hassan and Holt (2004) | P | Y | OS | Y | N |
| Holmes and Walker (2007) | P | Y | Toy | Y | N |
| Huang and Song (2007) | Y | Y | Toy | P | N |
| Ishio et al. (2004) | P | Y | OS | N | N |
| Jász et al. (2008) | P | Y | OS | N | P |
| Jiang et al. (2008) | P | P | OS | N | N |
| Kagdi and Maletic (2007) | Y | Y | None | N | N |
| Law and Rothermel (2003b) | Y | Y | Industrial | N | P |
| Leitch and Stroulia (2003) | P | Y | Industrial | N | P |
| Lienhard et al. (2007) | Y | Y | OS | N | N |
| Maule et al. (2008) | P | Y | Industrial | Y | P |
| Moise and Wong (2005) | Y | Y | OS | P | P |
| Nagappan and Ball (2007) | Y | Y | Industrial | Y | P |
| Pfaltz (2006) | Y | P | OS | N | N |
| Robillard (2008) | P | Y | OS | N | N |
| Ronen et al. (2006) | P | P | Industrial | P | P |
| Sangal et al. (2005) | Y | P | OS | P | P |
| Tallam and Gupta (2007) | Y | P | OS | N | N |
| Xiao and Tzerpos (2005) | P | P | OS | N | Y |
| Xin and Zhang (2007) | Y | P | Toy | N | N |
| Zhang and Ryder (2007) | Y | Y | OS | P | N |
| Zhao (2002) | N | P | Toy | N | N |
| Zimmermann and Nagappan (2007) | Y | Y | Industrial | Y | P |
| Zimmermann and Nagappan (2008) | Y | Y | Industrial | N | N |

**Table 8** Assessment of studies using description and model-based information

| Reference | Definition | Application area | Case | Easy to use | Ready to use |
|---|---|---|---|---|---|
| Garousi et al. (2006) | Y | P | Industrial | P | N |
| Ivkovic and Kontogiannis (2006) | N | Y | Toy | N | N |
| Jourdan et al. (2006) | Y | Y | Toy | N | N |
| Khan et al. (2008) | Y | Y | Industrial | Y | P |
| Korel et al. (2002) | Y | Y | Toy | P | Y |
| Liangli et al. (2006) | N | P | None | N | N |
| Mao et al. (2007) | N | Y | None | N | N |
| McComb et al. (2002) | Y | N | Industrial | N | N |
| Moraes et al. (2005) | Y | Y | OS | N | N |
| Ryser and Glinz (2000) | Y | Y | Other | N | N |
| Stafford et al. (2003) | P | P | None | P | N |
| Stafford and Wolf (2001) | Y | P | Toy | N | N |
| Vasilache and Tanaka (2005) | P | N | None | N | N |
| Vieira et al. (2001) | Y | P | Toy | N | N |
| Vieira and Richardson (2002) | Y | P | None | P | N |
| Xing and Stroulia (2006) | P | P | OS | Y | P |
| Zhao (2001) | Y | Y | None | N | N |

practitioners' perception. From the literature, we distinguish the type of system used for the validation of the proposed solution: Industrial, Open Source (OS), Toy (typically a very small piece of software not available on any open source website such as Sourceforge), Other (typically an experiment showing proof of the concept), or None. From the practitioners' perception (see Sections 7.2, 7.3 and 7.4), we distinguish whether the solution is easy to use and ready to use in our context. For Easy to Use, Y (yes) means that the overhead to identify dependencies is negligible; P (partly) means that we can cope with the overhead and required resources without considerable effort; N (no) means that there are major concerns about overhead and required resources. For Ready to

**Table 9** Assessment of studies using configuration and run-time monitored information

| Reference | Definition | Application area | Case | Easy to use | Ready to use |
|---|---|---|---|---|---|
| Agarwal et al. (2004) | Y | Y | Other | N | N |
| Callo Arias et al. (2008) | Y | P | Industrial | P | P |
| Brown et al. (2001) | P | Y | Other | N | N |
| Gao et al. (2004) | N | Y | Toy | N | N |
| Gupta et al. (2003) | Y | Y | Other | N | N |
| Keller et al. (2000) | Y | Y | Industrial | P | P |
| Keller and Kar (2000) | Y | Y | Industrial | P | P |
| Li et al. (2005b) | N | P | Toy | N | N |
| Narayanasamy (2006) | Y | Y | Other | N | N |
| Steinle et al. (2006) | P | Y | Industrial | N | N |
| Xiao and Urban (2008) | Y | Y | Other | N | N |

Use, Y (yes) means that tool-support and a well defined application process are readily available for the solution (e.g. a tool can be downloaded from a public website); P (partly) means that we can implement tool-support and application process of the solution without considerable effort; N (no) means that there are major concern about identifying and implementing the tool-support and application process for the solution.

## 7.1 Applicability of Definition of Dependencies

Although the literature provides various conceptual definitions, we observed that they do not immediately capture what is considered a dependency in practice. If we look back at our first research question: *What are the proposed definitions of dependencies?* and then ask *do they match the practice?*, we identified that the answer to the second part is 'no', at least not entirely. There are two major reasons for that: first, the definitions only concern specific types of dependencies and thus have a very narrow scope; second, they do not take into account the context of the system or the organization. We elaborate on these reasons in the next two paragraphs.

Practitioners would primarily find a definition useful if it applies to any kind of dependency: for example a dependency in the source code, a dependency which exists between components at run-time, or any other kind of dependency involving elements such as data and hardware resources (i.e. dependencies to the aggregate system). All of these aspects need to be studied because they influence the work of software developers'. Therefore the current definitions are not broad enough to encompass these different types of dependencies but they only deal with specific types.

The second concern of practitioners is that whether something is a dependency or not, depends on the situation. In practice, relationships among system elements are characterized as dependencies taking into account the system's characteristics, the development organization, and the development infrastructure. Thus, when development activities are distributed over multiple practitioners (e.g. architects, designers, and developers), and standard development environments (including compilers), support the various development activities, a call relationship between two functions can be considered as a dependency if the two sides of the relationship are managed by different practitioners. However, a similar relationship between components controlled by the same practitioner is usually not considered as a relevant dependency. In other words, as long as a practitioner can manage a relationship without the attention of more practitioners or additional tooling, such a relationship is usually not considered to be a relevant dependency.

To alleviate these two issues, we came up with a definition, based on the literature, our observations, and discussions with practitioners at Philips Healthcare. The definition, which resembles the definition described in De Souza (2005) is as follows:

In a large software-intensive system, a dependency is a relationship between two or more of the system's components or with the aggregate system. It causes that, when one of the involved components changes, the development organization needs to make considerable changes in one or more of the related components or the aggregate system.

7.2 Source Code-based Solutions

### 7.2.1 Static Analysis

Source code-based solutions that use static analysis are usually meant for identifying structural dependencies. As structural dependencies are implemented through code constructs such as function calls and shared variables, approaches that use this information have a high degree of accuracy when it comes to the dependencies they identify.

Furthermore, as the dependencies relate to tangible, explicit constructs that one can point out directly in the source code, analyzing and resolving problems with them is a relatively straightforward activity for practitioners compared to solutions described in Sections 6.2 and 6.3. In the latter two cases, it is usually necessary to "translate" the high-level dependencies to the underlying code constructs. This makes source code-based solutions very pleasant for practitioners (e.g., programmers and designers) and partly explains the appeal these solutions have on them.

In practice, however, there are concerns about the scalability and heterogeneity of solutions using static analysis. The first concern is about the enormous volume of information (in the form of dependency graphs) that is generated by these solutions. Thus a considerable effort is need to extract or identify actual dependencies. Part of this problem is caused by the low granularity used by most static analysis methods, which does not align with the high-level perspective of practitioners like software architects and managers. A second concern is that currently most static analysis solutions are not designed to cross the boundaries between software components implemented with different programming languages. This problem is even bigger when the implementation use languages with different paradigms (e.g. procedural and object-oriented) and off-the-shelf components whose source code is not available.

There is some room for improvement for the scalability and heterogeneity of static analysis solutions. For example, heterogeneity is addressed in Kontogiannis et al. (2006) for large multi-language software systems. However, Moise and Wong (2005) and Cossette and Walker (2007) showed that mechanisms and extensions of static cross-language analysis are still expensive and miss dependencies due to dynamic behavior. Thus, static analysis solutions do need to be complemented at least with dynamic analysis solutions. In practice, we observed that such combination will be need to analyze the role of source code artifacts in the following type of relationships. First, some relationships arise due to differences between implementation, deployment, and the actual system platform. For example, the actual code components that are deployed and used during execution could be different from, but compatible with, the ones used in the implementation and build time. Second, other relationships arise due to communication and interactions between software and hardware elements, which are relevant to manage the performance and availability of a system.

### 7.2.2 Dynamic Analysis

Dynamic analysis solutions share the properties of and complement static analysis solutions. In addition, dynamic analysis solutions have two main advantages over static analysis solutions. First, they enable the identification of relationships between code artifacts that only happen at runtime. Second, their sources of information,

i.e., execution traces do not depend on the specific syntax or semantic of individual programming languages, which makes the application of dynamic analysis solutions easier when working with a systems with heterogeneous implementation.

In practice, however, we observed that there are two main aspects that practitioners concern about working with dynamic analysis solutions. First, we need to balance the overhead produced by the techniques that are used to collect execution traces, e.g., source code instrumentation, platform profiling, and compiler profiling. The application of these techniques to a system with heterogeneous and large source code repository, and with components with partial or no source code available produces changes in the original runtime behavior of the system. These changes include variations of end-user-visible metrics such as performance and reliability, which are important qualities of large and complex system.

Second, we need to choose and plan the instrumentation level to mange the overhead, but also the amount of execution traces. If the instrumentation is done at the level of functions, methods, or other fine grain code elements the amount of execution traces will be large. Then, processing this fine grain data to extract useful high-level information will be difficult. This situation is a well know issue, which can be addressed with summarizing and visualization techniques (Cornelissen et al. 2007; Safyallah and Sartipi 2006).

### 7.2.3 Historical Analysis

Historical analysis solutions are a bit different from static and dynamic analysis solutions. Historical analysis solutions happen to identify dependencies that are not based on explicit source code constructs. Nevertheless, these dependencies are interesting for practitioners because they can show relationships between elements that are managed by different teams or organizations. In Section 7.1 we described that practitioners are interested in dependencies that require them to make changes in different parts of the system. By analyzing historical data, it is possible to identify files that changed together in the past and may change in the future irrespective of how the dependency has been implemented.

In practice, however, the availability and quality of historical data play an important role in how well historical analysis solutions identify useful relationships. We observed that historical data can be easily discarded due to changes or restructures in the source code archive. Thus, relationships being identified based on previous historical data will not longer exist.

### 7.2.4 Semantic Analysis

Recently, the semantics of source code has been used to identify related parts of in the software archive. By looking at the frequencies of words in files, classes, or functions relations between files can be uncovered (Kuhn et al. 2007; Marcus and Poshyvanyk 2005; Marcus et al. 2005; Spek et al. 2008). Similar to relationships identified by historical analysis, relationships identified in this way are not directly based on source code constructs. Relationships identified in this way, can point to related parts in the source code even when there is no structural dependency present. Similar to historical analysis solutions, this should facilitate identifying relevant dependencies irrespective of the underlying implementation. However, as there is

currently only limited practical experience with these approaches, their practical applicability is subject for further study.

### 7.2.5 Applicability in Practice

In short, source-code based solutions identify dependencies through code constructs such as function calls and shared variables. Approaches that use this concrete evidence have a high degree of accuracy when it comes to the dependencies they identify, which makes very reliable and very attractive for practitioners as the resulting information is very tangible. These properties make these techniques especially well suited for, amongst other things, predicting the impact of changes, refactoring and feature analysis as can be seen from Table 4. However, due to their nature they are less suited to analyze the runtime system behavior. Furthermore, there is still some research necessary in order to handle the massive amount of data obtained from large software-intensive systems and cross the borders of heterogeneous implementations.

## 7.3 Model-based Solutions

### 7.3.1 Diagrammatic Descriptions

Diagrammatic descriptions such as design models and architectural views are the source of information for architecture-driven approaches. Practitioners i.e. software architects and designers, like to facilitate the communication between them during development projects by means of these descriptions. We observed that conducting dependency analysis using design models and documents helps to identify dependencies between components. For instance, when phasing-out a legacy component, architects and designers analyze design documentation to plan the phasing-out process. Practitioners try to identify all of the dependencies between the component and the rest of the system. Once the dependencies are identified, some design documents may be created to describe what the dependencies and how to remove them. Later on, during the execution of the phasing-out process, developers interact with designer and follow the design documents to implement the actual changes in the system.

In practice, however, we observed that using design models and architectural descriptions to identify dependencies is not a straightforward activity due to one important factor: keeping the models and documents up-to-date. Keeping design models and architectural descriptions up-to-date and synchronized with the implementation is often a deficient activity. We observed that additional effort is needed to update and create models. Practitioners often need to reverse engineer the system realization by consulting experts and studying the system's source code. This activity is often hard to do with large and heterogeneous implementations maintained in a geographically spread organization operating in different time zones.

### 7.3.2 Semi-formal Descriptions

ADLs (Stafford and Wolf 2001; Zhao 2001) and EFSM (Jourdan et al. 2006; Korel et al. 2002) are some of the tools that have emerged as potential solutions for formally describing system architectures at a higher level of abstraction. These initiatives provide support for modeling not just the system structure, but also the behavior

and communication between components. Thus, we consider that the introduction of these initiatives to architecture description can ease architectural analysis, i.e. dependency analysis at an early stage.

However, our observation in practice is that practitioners, i.e. architects and designers, often use informal diagrammatic representations rather than semi-formal descriptions such as ADLs or EFSMs. The reasons for this situation and which should be addressed include the lack of knowledge dissemination (e.g. industrial experiences), tool support to use these semi-formal language descriptions, and support to link semi-formal descriptions to implementation.

### 7.3.3 Traceability

New requirements often trigger changes among software elements, such as architecture, design, and implementation. Thus, two-way traceability between requirements and the software architecture elements satisfying the requirements is needed, especially to keep architecture and design descriptions updated and inline with the new requirements as well as with the implementation. Solutions that identify traceability dependencies (Egyed 2003) represent alternatives to improve the synchronization between design and implementation. These solutions can improve the use of diagrammatic and semi-formal description for dependency analysis. However, since traceability analysis uses code or execution-based information, its application should take into account our observations in Section 7.2.

### 7.3.4 Applicability in Practice

In brief, solutions using diagrammatic and semi-formal descriptions are more appealing for practitioners following architecture-driven approaches. The Practitioners find useful these solutions for the abstraction level and support to describe dependency information at an architecture level. However, for an efficient application of these solutions, we need to keep up-to-date and synchronize the system requirements, design, and implementation. Thus, dependency analysis solutions will need to address these factors before hands. For example, we need solutions that improve the accessibility and validity of architecture and design descriptions taking into account the size of development organization, constant system evolution, distributed development locations, extensive documentation, and the mental models of practitioners.

### 7.4 Run-time Monitored and Configuration-based Solutions

Solutions using run-time monitored and configuration data have considerable contribution to support architecture-driven processes. These solutions facilitate the understanding of the system execution at a system level, including software and hardware components. Dependency models at this level are easy to integrate into the system documentation, and reusable in further dependency analysis activities (Brown et al. 2001). In practice, we observed three main aspects that support their application. First, these solutions are not limited by implementation technology borders. Second, the techniques used by these solutions to collect runtime data are considered less intrusive. And third, these solutions identify dependencies without requiring access to the system source code. Nevertheless, the identified dependencies are limited to those monitored scenarios and the components whose configuration was analyzed.

However, we observed that the analysis of major system components as black boxes needs to be adjusted for dependency analysis within the development cycle of large systems. Often in a development cycle, changes in the implementation may eventually cause undesired variations of end-user-visible properties. Thus to tune variations, we need to improve the transparency of these solutions. Thus, we can be able to trace back the change and identify the faulty element (e.g. running application, object, class, or method).

### 7.4.1 Combining Information Sources

Combining sources of information is an approach that source code-based solutions (Section 6.1) use as a way for coping with incompleteness of independent solutions. We observed that combining different sorts of run-time monitored data like logging and process activity,[2] helps to link system run-time components (seen as black boxes) to source code related artifacts, e.g., binaries and libraries.

### 7.4.2 Applicability in Practice

In brief, we found that solutions using run-time monitored and configuration information are used and applicable in practice due to two main characteristics. First, practitioner highlight that these solutions are non-intrusive with respect to the development activities. Often, in a research setting, the overhead and maintenance cost of an infrastructure to collect data for dependency analysis is overlooked. In contrast, practitioners are more concerned about the cost and overhead of maintaining a reliable and up-to-date instrumentation of their system. This is even more important, in heterogeneous situations where multi-vendor components are used and instrumentation cannot be inserted into the system for security, licensing, lack of knowledge, or other technical constraints.

Second, although these solutions are limited by their coverage and links to the system source code, practitioners consider these solutions valid approximations, especially for problem-driven approaches. In practice, the analysis is restricted to a representative set of execution scenarios and additional solutions are available to provide links to source code artifacts (see Section 6.1).

## 8 Threats to Validity

In this section we discuss the threats to the validity of the study, in terms of internal validity (the validity of the actual review) and external validity (the generalizability of the results for other domains).

8.1 Internal Validity

### 8.1.1 About the Study Search

*Search Strategy*    At the time we began the review, we had a limited knowledge about dependency analysis research and related set of venues where we could conduct

---

[2]This reference has been removed in order to hide personal information of the authors.

a manual search. Thus, we decided for an automatic keyword search strategy with Google Scholar, considering that practitioners could easily replicate it and we could find relevant papers in a short time.

Although systematic reviews often involve automatic keyword searches, Brereton et al. (2007) pointed out that: "Current software engineering search engines are not designed to support systematic literature reviews. Unlike medical researchers, software engineering researchers need to perform resource-dependent searches". Thus, our choice of an automatic keyword search strategy is a threat to validity for the completeness of our search results.

A manual search on specific venues is another alternative for study search. This alternative is consistent with the practices of other researchers looking at research trends (Cornelissen et al. 2009; Kitchenham et al. 2009). However, manual search process of a specific set of venues often implies missing relevant studies from excluded venues (Kitchenham et al. 2009). Based on the number of venues in Table 2, we consider that identifying a representative set of venues related to dependency analysis is hard, especially without prior experience or the advice of experts in the topic.

The effectiveness of both automatic keyword and manual searches is arguable for search studies. To evaluate the implications of a manual search, we provide Table 2 with a representative set of venues. It would be of great interest to take this set of venues as a reference, in order to replicate our search study for dependency analysis and compare the results.

*Google Scholar* Another threat to validity for our search study is the choice for Google Scholar as the search engine. This engine has both advantages and drawbacks. Google Scholar is easily accessible and often the first choice for practitioners when looking for a solution to a problem they are faced with. This search engine offers a search experience familiar to anyone with even limited exposure to Google (Robinson and Wuseman 2007). This search engine allowed us to search across different digital libraries (e.g., IEEEXplore and ACM Portal) and cove various venues (see Table 2). Nevertheless, the software of Google Scholar has poor performance with the highly structured and tagged scholarly documents. It still has serious deficiencies with basic search operations, and does not have any sort options (beyond the questionable relevance ranking) (Jacsó 2008). Also, it offers filtering features by data elements, which are present only in a very small fraction of the records (such as broad subject categories) and/or are often absent and incorrect in Google Scholar even if they are present correctly in the source items (Jacsó 2008). A more thorough review of the advantages and disadvantages of Google Scholar can be found in Jacsó (2008) and Lewandowski (2010).

There are two alternatives to Google Scholar. The first is using the various search engines of digital libraries. However, one would need to identify a representative set of digital libraries and cope with the limitations in the search engines of those electronic libraries (e.g. ACM Portal does not support complex logical combination (Brereton et al. 2007; Kitchenham 2004b)). The second is using search engines of dedicated databases of research literature. For example, Scopus[3] is an option that

---

[3]http://www.scopus.com/

could support our search queries and perhaps provide less false positives. However, engines of this type are not yet known and accessible to most practitioners.

### 8.1.2 About the Study Selection

Through the study selection process, we noticed the risk of bias due to our preference for solutions related to our individual research interests and time constraints. To reduce bias due to our research preferences, we decided to pay attention and discuss papers in conflict (see Pilot Selection in Section 3.1.2), but still this process only involved two reviewers. An alternative could be the inclusion of a third-party reviewer with a particular or broad research interest about dependency analysis.

Another threat for the study selection was constraints in time and effort: we decide not to extend the study search during the process since we need to deliver results to the practitioners. We could have extended the search and selection with references in the selected papers and include gray literature sources such as PhD theses and technical reports. It could have helped us to find more relevant studies, which could influence the practitioners' perception and the results of the review. But due to the practical nature of our project, we could not exceed our time budget.

### 8.1.3 About the Quality Assessment

Our decisions for the quality criteria definition and evaluation were based on our judgment on the content of the selected papers. The given scores aim at illustrating the basis of our judgment, but a quantitative analysis on them could provide explicit evidence to support our decision. A quantitative analysis would be useful to investigate relationships between quality factors (Kitchenham et al. 2009) or characterizations to identify research trends and avenues for future research (Cornelissen et al. 2009). We consider this as promising future work.

### 8.1.4 About Data Extraction and Data Synthesis

The data extraction and data synthesis in our study were driven by the goal and context of the review. A validity threat is that we extracted the data working on two different sets of papers without directly checking the extraction. To mitigate this threat, we checked and improved the extracted data afterwards by working together in two occasions. First, during the synthesis process where we merged the extracted data to construct the summaries. Second, interacting with practitioners to adjust the structure and content of the summaries. Although this process gave us confidence about the quality of the data extraction and data synthesis process, similar quality could be achieved with less effort following a extractor/checker mode (Kitchenham et al. 2009) early during the data extraction phase.

### 8.2 External Validity

The information in the study is based on research results from the literature according to the perception and judgment of a number of practitioners in Philips MRI. Thus, a potential threat to validity is that the information in the overview is specific to our research context and cannot be considered generalized to other organizations or domains.

However, we argue that the MRI system is representative of large and complex systems, and that the MRI development organization is characteristic of other organizations. In fact, there are many other organizations developing similar large and complex systems. A modern car, for instance, may contain 20–30 microprocessors, with software controlling aspects such as engine ignition, pollution control, security, air-conditioning, car radio, or even the seat position (Stevens 1998). Another example of a complex system are modern televisions (van Ommering 2002), which consist of mechanics, electro-optics, electronics, and software. It typically takes 100 software engineers 2 years to build the software for a high-end television. Many more comparable complex systems exist, such as cellular phones (Neuvo 2004), copiers, wafer steppers used in the semiconductor industry, and airplanes (Kossiakoff and Sweet 2002). In this sense, the study is a representative reference for other practitioners and researchers working with dependency analysis, similar development organizations, and similar types of systems.

## 9 Concluding Remarks

We have followed the guidelines for systematic review to present research results about dependency analysis and to assess their value. The results of this review describes the match between research and practice as elaborated throughout the answers to our research questions that build the overview. It has helped us to identify opportunities to improve dependency analysis as part of the development of the Philips MRI system.

Researchers and practitioners can use the overview to learn about the state-of-art in dependency analysis and how it matches the characteristics and development of a representative large and complex software-intensive system. Researchers can take into account the research opportunities and findings described in Section 7. Furthermore, the information in the overview can be used to identify trends in research practices, e.g., to identify the usual venues for dependency analysis, the application areas with more attention, the most popular sources of information, and the usual validation.

Finally, the conduction of this review is our attempt to match an existing methodology to a particular problem domain (Glass 2004). In this sense, researchers and practitioners can take this review as a reference in three contexts. First is for researchers and practitioners working in the same domain, who can match their own practice to the presented results. Second is to construct a similar overview of dependency analysis matching the development and characteristic of systems in other domains, e.g., enterprise software systems. Third is to construct similar overviews about other software engineering methods or techniques for systems in the domain of the MRI system, as well as, for systems in other domains.

Finally, we consider that other researchers can use the practice-driven approach of this review to evaluate other types of solutions and that it not only describes practical issues, but also exposes research opportunities that can have a strong impact on the way that software organizations develop large and complex software-intensive systems.

# References

Agarwal MK, Appleby K, Gupta M, Kar G, Neogi A, Sailer A (2004) Problem determination using dependency graphs and run-time behavior models. In: Sahai A, Wu F (eds) 15th IFIP/IEEE international workshop on distributed systems: operations and management. Springer, pp 171–182

Allen R, Garlan D (1997) A formal basis for architectural connection. ACM Trans Softw Eng Methodol 6(3):213–249. doi:10.1145/258077.258078

Alzamil Z (2007) Redundant coupling detection using dynamic dependence analysis. In: International conference on software engineering advances (ICSEA 2007)

Balmas F, Wertz H, Chaabane R, Artificielle L (2005) Visualizing dynamic data dependences as a help to maintain programs. In: International conference on software maintenance

Binkley D, Harman M 2005 Locating dependence clusters and dependence pollution. In: 21st IEEE international conference on software maintenance, pp 177–186

Bohnet J, Voigt S, Döllner J (2009) Projecting code changes onto execution traces to support localization of recently introduced bugs. In: ACM symposium on applied computing. ACM, New York, pp 438–442

Breivold HP, Crnkovic I, Land R, Larsson S (2008) Using dependency model to support software architecture evolution. In: 23rd IEEE/ACM international conference on automated software engineering-workshops, ASE workshops, pp 82–91

Brereton P, Kitchenham BA, Budgen D, Turner M, Khalil M (2007) Lessons from applying the systematic literature review process within the software engineering domain. J Syst Softw 80(4):571–583. doi:10.1016/j.jss.2006.07.009

Brown A, Kar G, Keller A (2001) An active approach to characterizing dynamic dependencies forproblem determination in a distributed environment. In: IEEE/IFIP international symposium on integrated network management proceedings, pp 377–390

Callo Arias TB, Avgeriou P, America P (2008) Analyzing the actual execution of a large software-intensive system for determining dependencies. In: Proceedings of the 2008 15th working conference on reverse engineering, pp 49–58

Cataldo M, Herbsleb JD, Carley KM (2008) Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: 2nd international symposium on empirical software engineering and measurement. ACM, Kaiserslautern, pp 2–11

Chen K, Rajlich V (2000) Case study of feature location using dependence graph. In: 8th international workshop on program comprehension. IEEE Computer Society, Washington, DC, p 241

Chen Z, Xu B, Yang H, Liu K, Zhang J (2000) An approach to analyzing dependency of concurrent programs. In: The first Asia-Pacific conference on quality software (APAQS'00), vol 3, p 12

Chen Z, Xu B, Zhao J (2002) An overview of methods for dependence analysis of concurrent programs. ACM Sigplan Not 37:45–52

Cornelissen B, Holten D, Zaidman A, Moonen L, van Wijk JJ, van Deursen A (2007) Understanding execution traces using massive sequence and circular bundle views. In: ICPC'07. IEEE Computer Society, Washington, DC, pp 49–58. doi:10.1109/ICPC.2007.39

Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R (2009) A systematic survey of program comprehension through dynamic analysis. Tech. rep. doi:10.1109/TSE.2009.28

Cossette B, Walker R (2007) Polylingual dependency analysis using island grammars: a cost versus accuracy evaluation. In: IEEE international conference on software maintenance. ICSM 2007, pp 214–223

Cox L, Delugach H, Skipper D (2001) Dependency analysis using conceptual graphs. In: Proceedings of the 9th international conference on conceptual structures, ICCS, pp 117–130

De Souza C (2005) On the relationship between software dependencies and coordination: field studies and tool support

Dong X, Godfrey M (2007) System-level usage dependency analysis of object-oriented systems. In: Proc. of the intl. conference on software maintenance, pp 375–384

Egyed A (2003) A scenario-driven approach to trace dependency analysis. IEEE Trans Softw Eng 29:116–132

Eisenbarth T, Koschke R, Simon D (2001) Aiding program comprehension by static and dynamic feature analysis. In: Proceedings of the international conference on software maintenance, pp 602–611

Eisenbarth T, Koschke R, Simon D (2003) Locating features in source code. IEEE Trans Softw Eng 29:210–224

Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. ACM Trans Program Lang Syst 9(3):319–349. doi:10.1145/24039.24041

Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley, Boston

Fundel K, Kuffner R, Zimmer R (2007) RelEx–relation extraction using dependency parse trees. Bioinformatics 23(3):365

Gao J, Kar G, Kermani P (2004) Approaches to building self healing systems using dependency analysis. In: IEEE/IFIP network operations and management symposium, NOMS 2004, vol 1, pp 119–132. doi:10.1109/NOMS.2004.1317649

Garousi V, Briand L, Labiche Y (2006) Analysis and visualization of behavioral dependencies among distributed objects based on UML models. In: Nierstrasz O et al (eds) International conference on model driven engineering languages and systems. Springer

Glass RL (2004) Matching methodology to problem domain. Commun ACM 47(5):19–21. doi:10.1145/986213.986228

Glorie M, Zaidman A, Van Deursen A, Hofland L (2009) Splitting a large software repository for easing future software evolution—an industrial experience report. J Softw Maint Evol Res Pract 21(2):113–141

Gotel O, Finkelstein A (1994) An analysis of the requirements traceability problem. In: Proc. first int'l conf. requirements eng., pp 94–101

Gupta M, Neogi A, Agarwal M, Kar G (2003) Discovering dynamic dependencies in enterprise environments for problem determination. In: Brunner M, Keller A (eds) 14th IFIP/IEEE international workshop on distributed systems: operations and management

Hamou-Lhadj A, Lethbridge TC (2004) A survey of trace exploration tools and techniques. In: CASCON'04. IBM Press, pp 42–55

Hassan A, Holt R (2004) Predicting change propagation in software systems. In: Proceedings of 26th international conference on software maintenance (ICSM'04)

Holmes R, Walker R (2007) Task-specific source code dependency investigation. In: 4th IEEE international workshop on visualizing software for understanding and analysis, 2007. VISSOFT 2007, pp 100–107

Huang L, Song Y (2007) Precise dynamic impact analysis with dependency analysis for object-oriented programs. In: Fifth international conference on software engineering research, management and applications, pp 374–381. doi:10.1109/SERA.2007.46

Ishio T, Kusumoto S, Inoue K (2004) Debugging support for aspect-oriented program based on program slicing and call graph. In: Proc. 20th IEEE international conference on software maintenance, pp 178–187

Ivkovic I, Kontogiannis K (2006) Towards automatic establishment of model dependencies using formal concept analysis. Int J Softw Eng Knowl Eng 16:499–522

Jacsó P (2008) Google scholar revisited. Online Inf Rev 32(1):102–114

Jász J, Beszédes A, Gyimóthy T, Rajlich V (2008) Static execute after/before as a replacement of traditional software dependencies. In: IEEE international conference on software maintenance. ICSM 2008, pp 137–146

Jiang T, Gold N, Harman M, Li Z (2008) Locating dependence structures using search-based slicing. Inf Softw Technol 50:1189–1209. doi:10.1016/j.infsof.2007.11.001

Jourdan Gv, Ritthiruangdech P, Ural H (2006) Test suite reduction based on dependence analysis. In: Al ALE (ed) 21th international symposium computer and information sciences. Springer, pp 1021–1030

Kagdi H, Maletic J (2007) Combining single-version and evolutionary dependencies for software-change prediction. In: Fourth international workshop on mining software repositories. IEEE Computer Society , p 17

Keller A, Blumenthal U, Kar G (2000) Classification and computation of dependencies for distributed management. In: Fifth international conference on computers and communications (ISCC 2000)

Keller A, Kar G (2000) Dynamic dependencies in application service management. In: International conference on parallel and distributed processing techniques and applications

Khan S, Greenwood P, Garcia A, Rashid A (2008) On the impact of evolving requirements-architecture dependencies: an exploratory study. In: Bellahsène Z, Léonard M (eds) 20th international conference advanced information systems engineering, pp 243–257

Kitchenham B (2004a) Procedures for performing systematic reviews. Tech. rep., Joint Technical Report Keele University TR/SE0104 and NICTA 0400011T.1

Kitchenham B (2004b) Software productivity measurement using multiple size measures. IEEE Trans Softw Eng 30(12):1023–1035. doi:10.1109/TSE.2004.104. Member-Mendes, Emilia

Kitchenham B, Brereton OP, Budgen D, Turner M, Bailey J, Linkman S (2009) Systematic literature reviews in software engineering—a systematic literature review. Inf Softw Technol 51(1):7–15. doi:10.1016/j.infsof.2008.09.009

Kontogiannis K, Linos P, Wong K (2006) Comprehension and maintenance of large-scale multi-language software applications. In: ICSM '06. IEEE Computer Society, Washington, DC, pp 497–500. doi:10.1109/ICSM.2006.20

Korel B, Tahat L, Vaysburg B (2002) Model based regression test reduction using dependence analysis. In: Proceedings. International conference on software maintenance, pp 214–223

Kossiakoff A, Sweet WN (2002) Systems engineering: principles and practice. Wiley-Interscience

Kuhn A, Ducasse S, Gîrba T (2007) Semantic clustering: identifying topics in source code. Inf Softw Technol 49(3):230–243. doi:10.1016/j.infsof.2006.10.017

Law J, Rothermel G (2003a) Incremental dynamic impact analysis for evolving software systems. In: Proceedings of the international symposium on software reliability engineering, pp 430–441

Law J, Rothermel G (2003b) Whole program path-based dynamic impact analysis. In: 25th international conference on software engineering. IEEE Computer Society, Washington, DC, pp 308–318

Leitch R, Stroulia E (2003) Assessing the maintainability benefits of design restructuring using dependency analysis. In: Proc. of the 9th international symposium on software metrics, pp 309–322

Lewandowski D (2010) Google scholar as a tool for discovering journal articles in library and information science. Online Inf Rev 34(2):250–262. doi:10.1108/14684521011036972

Li B, Zhou Y, Wang Y, Mo J (2005a) Matrix-based component dependence representation and its applications in software quality assurance. ACM SIGPLAN Not 40:36

Li Y, Zhang M, Hou C (2005b) An active method to building dynamic dependency model for distributed components. In: ICAC '05: Proceedings of the second international conference on automatic computing. IEEE Computer Society, Washington, DC, pp 337–338. doi:10.1109/ICAC.2005.7

Liangli M, Houxiang W, Yansheng L (2006) The design of dependency relationships matrix to improve the testability of component-based software. In: Sixth international conference on quality software. QSIC 2006, pp 93–98

Lienhard A, Greevy O, Nierstrasz O (2007) Tracking objects to detect feature dependencies. In: 15th IEEE international conference on program comprehension, pp 59–68

Lind-Nielsen J, Andersen H, Hulgaard H, Behrmann G, Kristoffersen K, Larsen K (2001) Verification of large state/event systems using compositionality and dependency analysis. Form Methods Syst Des 18(1):5–23

Loyall JP, Mathisen SA (1993) Using dependence analysis to support the software maintenance process. In: ICSM '93. IEEE Computer Society, Washington, DC, pp 282–291

Mao C, Zhang J, Lu Y (2007) Using dependence matrix to support change impact analysis for CBS. In: Fifth international conference on computational science and applications (ICCSA 2007), pp 192–200. doi:10.1109/ICCSA.2007.25

Marcus A, Poshyvanyk D (2005) The conceptual cohesion of classes. In: ICSM '05. IEEE Computer Society, Washington, DC, pp 133–142. doi:10.1109/ICSM.2005.89

Marcus A, Rajlich V, Buchta J, Petrenko M, Sergeyev A (2005) Static techniques for concept location in object-oriented code. In: IWPC '05, pp 33–42

Maule A, Emmerich W, Rosenblum D (2008) Impact analysis of database schema changes. In: 30th international conference on software engineering, pp 451–460

McComb D, Robe S, Hoare S, Crawford-Hines S (2002) Dependency analysis and visualization as tools to prolong system life. In: Proceedings of the 26th annual international computer software and applications conference (COMPSAC'02), pp 463–465

Mehta NR, Medvidovic N, Phadke S (2000) Towards a taxonomy of software connectors. In: ICSE '00. IEEE Computer Society, Los Alamitos, p 178. doi:10.1109/ICSE.2000.10009

Moise D, Wong K (2005) Extracting and representing cross-language dependencies in diverse software systems. In: WCRE '05, Washington, DC, USA: IEEE Computer Society, pp 209–218

Moraes RLO, Martins E, Mendes NV (2005) Fault injection approach based on dependence analysis. In: COMPSAC'05. IEEE Computer Society, Washington, DC, pp 181–188. doi:10.1109/COMPSAC.2005.78

Moriconi M, Winkler TC (1990) Approximate reasoning about the semantic effects of program changes. IEEE Trans Softw Eng 16(9):980–992

Myers G (1975) Reliable software through composite design. Petrocelli/Charter, New York

Nagappan N, Ball T (2007) Using software dependencies and churn metrics to predict field failures: an empirical case study. In: First international symposium on empirical software engineering and measurement, ESEM 2007, pp 364–373

Narayanasamy S, Pereira C, Calder B (2006) Recording shared memory dependencies using strata. In: Proceedings of the 12th international conference on architectural support for programming languages and operating systems. ACM, p 240

Neuvo Y (2004) Cellular phones as embedded systems. In: IEEE international solid-state circuits conference, pp 32–37

Pfaltz JL (2006) Using concept lattices to uncover causal dependencies in software. In: Missaoui R, Schmid J (eds) 4th international conference formal concept analysis, vol 3874. Springer, p 233

Podgurski A, Clarke LA (1990) A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans Softw Eng 16(6):965–979

Robillard M (2008) Topology analysis of software dependencies. ACM Trans Softw Eng Methodol (TOSEM) 17:18. doi:10.1145/13487689.13487691

Robinson ML, Wusteman J (2007) Putting google scholar to the test: a preliminary study. Program: Electronic Library and Information Systems 41(1):71–80

Ronen I, Dor N, Porat S, Dubinsky Y (2006) Combined static and dynamic analysis for inferring program dependencies using a pattern language. In: Conference of the Center for Advanced Studies on Collaborative Research, p 3

Ryser J, Glinz M (2000) Using dependency charts to improve scenario-based testing. In: Proceedings of the 17th international conference on testing computer software (TCS2000)

Safyallah H, Sartipi K (2006) Dynamic analysis of software systems using execution pattern mining. In: ICPC '06. IEEE Computer Society, Washington, DC, pp 84–88. doi:10.1109/ICPC.2006.19

Sangal N, Jordan E, Sinha V, Jackson D (2005) Using dependency models to manage complex software architecture. In: Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. ACM, p 176

Spek Pvd, Klusener S, van de Laar P (2008) Towards recovering architectural concepts using latent semantic indexing. In: CSMR '08: Proceedings of the 2008 12th European conference on software maintenance and reengineering. IEEE Computer Society, Washington, DC, pp 253–257. doi:10.1109/CSMR.2008.4493321

Stafford JA, Wolf AL (2001) Architecture-level dependence analysis for software systems. Int J Softw Eng Knowl Eng 11:431–451

Stafford J, Wolf A, Caporuscio M (2003) The application of dependence analysis to software architecture descriptions. In: Bernardo M, Inverardi P (eds) Third international school on formal methods for the design of computer, communication and software systems: software architectures, pp 52–62

Stafford JA, Wolf AL (1998) Architecture-level dependence analysis in support of software maintenance. In: ISAW '98. ACM Press, New York, pp 129–132. doi:10.1145/288408.288441

Steinle M, Aberer K, Girdzijauskas S, Lovis C (2006) Mapping moving landscapes by mining mountains of logs: novel techniques for dependency model generation. In: Proceedings of the 32nd international conference on very large data bases. VLDB Endowment, p 1102

Stevens R (1998) Systems engineering: coping with complexity. Prentice Hall

Stevens W, Myers G, Constantine L (1974) Structured design. IBM Syst J 13(2):115–139

Tallam S, Gupta R (2007) Unified control flow and data dependence traces. ACM TACO 4(3). doi:10.1145/1275937.1275943

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: Evol'07, pp 48–53. Paris, France

van Ommering R (2002) Building product populations with software components. In: ICSE '02: Proceedings of the 24th international conference on software engineering. ACM, New York, pp 255–265. doi:10.1145/581339.581373

Vasilache S, Tanaka J (2005) Bridging the gap between analysis and design using dependency diagrams. In: Third ACIS int'l conference on software engineering research, management and applications (SERAŠ05)

Vieira M, Dias M, Richardson D (2001) Describing dependencies in component access points. In: Procs of the 23rd intern. conf. on software engineering

Vieira M, Richardson D (2002) Analyzing dependencies in large component-based systems. In: 17th IEEE international conference on automated software engineering, ASE 2002, pp 241–244

Watkins R, Neal M (1994) Why and how of requirements tracing. IEEE Softw 11(4):104–106. doi:10.1109/52.300100

Xiao C, Tzerpos V (2005) Software clustering based on dynamic dependencies. In: Proceedings of the ninth European conference on software maintenance and reengineering, pp 124–133

Xiao Y, Urban SD (2008) Recovery of concurrent processes in a service composition environment using data dependencies. In: Proceedings of the OTM 2008 confederated. Springer, pp 139–156

Xin B, Zhang X (2007) Efficient online detection of dynamic control dependence. In: Proceedings of the 2007 international symposium on software testing and analysis. ACM, p 195

Xing Z, Stroulia E (2006) Understanding the evolution and co-evolution of classes in object-oriented systems. Int J Softw Eng Knowl Eng 16:23–51

Zhang W, Ryder B (2007) Discovering accurate interclass test dependences. In: 7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering, pp, 55–61

Zhao J (2001) Using dependence analysis to support software architecture understanding. In: New technologies on computer software

Zhao J (2002) Change impact analysis for aspect-oriented software evolution. In: International workshop on principles of software evolution, pp 108–112

Zimmermann T, Nagappan N (2007) Predicting subsystem failures using dependency graph complexities. In: Proceedings of the The 18th IEEE international symposium on software reliability. IEEE Computer Society, pp 227–236

Zimmermann T, Nagappan N (2008) Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th international conference on software engineering, pp 531–540

**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 PhD candidate in the Software Engineering and Architecture Group of University of Groningen. His professional experience and interest include the architecture and design of software solutions for high-tech products, embedded systems, and distributed systems.

**Pieter van der Spek** received a master's degree in informatics from Delft University of Technology in 2004. After his master's degree, he worked for several years at a small consultancy firm and was responsible for design, development, and testing of various software products. In 2006, he became a PhD-researcher at the VU University Amsterdam working on methods to facilitate tasks related to maintaining software systems. In 2010, he received his PhD-degree at the VU University. He is presently working for a large technical IT service provider. His research interests include reverse engineering, software maintenance, and design of high-tech products and embedded systems.



**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.