Execution architecture views for evolving software-intensive systems

Callo Arias, Trosky Boris

Chapter 2

A Systematic Review of Dependency Analysis Solutions

Published as: Trosky B. Callo Arias, Pieter van der Spek, Paris Avgeriou – *"A practice-driven systematic review of dependency analysis solutions,"* Empirical Software Engineering, March 2011.

Abstract

When following architecture-driven strategies to develop large software-intensive systems, the analysis of the dependencies is not an easy task. In this chapter, 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.

2.1 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) scanner developed by our industrial partner, Philips Healthcare MRI.

In the context of our project, one of our early observations was that indeed information about dependencies was important to improve the evolvability of the Philips MRI scanner. However, we also observed that in the development of this system, dependency analysis was a time-consuming activity conducted on an ad hoc basis and without proper support (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 chapter, we report a systematic literature review (Kitchenham 2004b) that we conducted to build the overview of dependency analysis. The review is practice-driven due to three main aspects. 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 practitioners can use, rather than research trends upon which researchers can base future research. Third, the review includes an assessment of research results taking into account the practical characteristics of the software embedded in the Philips MRI scanner and its 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 chapter focuses primarily in supporting practitioners to learn about the state-of-art in dependency analysis. In addition, we show how the state-of-art matches the characteristics and development of a representative large and complex software-intensive system.

The remainder of this chapter is organized as follows. Section 2.2 describes the context in which we performed the review and the specific research questions. Section 2.3 describes the protocol of the review. Section 2.4 starts the overview describing conceptual aspects related to dependency analysis. Next, Section 2.5 describes the application areas where dependency analysis contribute to. Section 2.6 describes a set of existing dependency analysis solutions classified by their source of information. Section 2.7 concludes the overview describing how the existing definitions and solutions match the practical requirements of our particular context, including the identified opportunities for improvement. In Section 2.8, we discuss threats to validity for the review and the results. Finally, Section 2.9 provides some concluding remarks.

2.2 Context and Research Questions

We conducted the review as part of our research in the Darwin project (van de Laar et al. 2007). In this project, we focus on how to improve the evolvability of software-intensive systems studying a Magnetic Resonance Imaging (MRI) scanner. Thus, when mentioning *practitioners* or *in practice* we refer to the developers and the intrinsic development of the software embedded in the Philips MRI scanner 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.

2.2.1 The Software of the Philips MRI Scanner

In Section 1.2.1, we introduced the Philips MRI scanner as a representative software-intensive system. This system combines various hardware components with a fair amount of software components. Table 1.1 summarizes the complexity of this system and its development organization. The software system comprises several million lines of code written in nine different programming languages (heterogeneous implementation). In addition, 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 is the complexity of its development organization. A Software Architecture Team (SWAT) monitors the architecture-centric evolution of the software embedded in the Philips MRI scanner. 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 and internal and external teams of programmers. The development teams of the Philips MRI scanner are multidisciplinary and with competencies in areas such as physics, electronics, mechanics, material science, software engineering, and clinical science. These various teams are spread in different geographically locations with different time zones. 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. These characteristics of the system and its development pose special requirements for dependency analysis activities. For example, while architects and designers are mostly interested in high-level or architectural dependencies, programmers see dependencies in terms of source-code level constructs such as function calls.

2.2.2 Research Questions

Table 2.1 shows the set of research questions that this systematic review aims at answering. We have defined this set of questions trying to generalize the following observations that arose within our interaction with practitioners:

Definitions of dependency : 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. Among other things, we identified that the perception of what constitutes a dependency is part of this implicit knowledge. Definitions of dependency are provided in the literature, but 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.

The need for dependency analysis : Our interaction with practitioners started assuming that dependency analysis was useful. However, we soon realized that we needed explicit evidence to support 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.

Existing solutions : 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.

Applicability of existing solutions : 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.

Table 2.1: *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 in the development software-intensive systems that can be addressed using dependency analysis.
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.

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. In addition, the results of this review could 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.

2.3 Design of the Review

The protocol for conducting our review is based on the guidelines for systematic literature reviews as proposed in (Kitchenham 2004b). Figure 2.1 illustrates the main phases of the review protocol: study search and selection, data extraction, data synthesis, and interpretation. The last phase, interpretation, is an addition to the proposed guidelines (Kitchenham 2004b), which we conducted to assess the applicability of the study results taking the practitioners perspective as reference. In the rest of this section, we describe the motivation and settings for each of the phases in the protocol. The threats to the validity of this study are discussed in Section 2.8.

2.3.1 Study search and selection

In this phase, we focused on searching and selecting 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.

Study search

The searching 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 provided in Section 2.8. 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 10 years. We experimented with several 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`

[1] `http://scholar.google.com/`

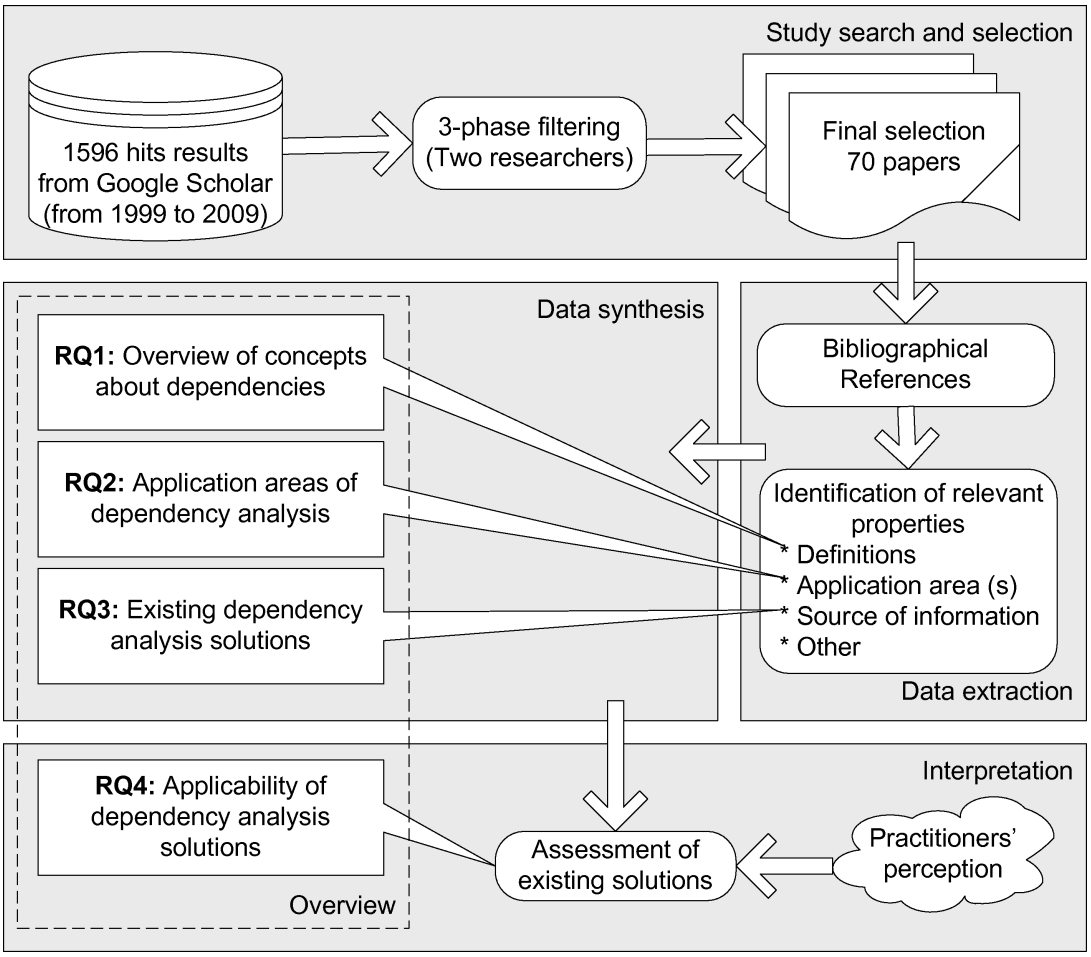

Figure 2.1: *Overview of the systematic review process.*

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 2.4.2, 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 that 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 an initial or pilot phase of the study. The third search string was designed taking into account the results of the pilot phase. Figure 2.2 zooms into the study search and selection phase of the review protocol.

The figure illustrates 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.

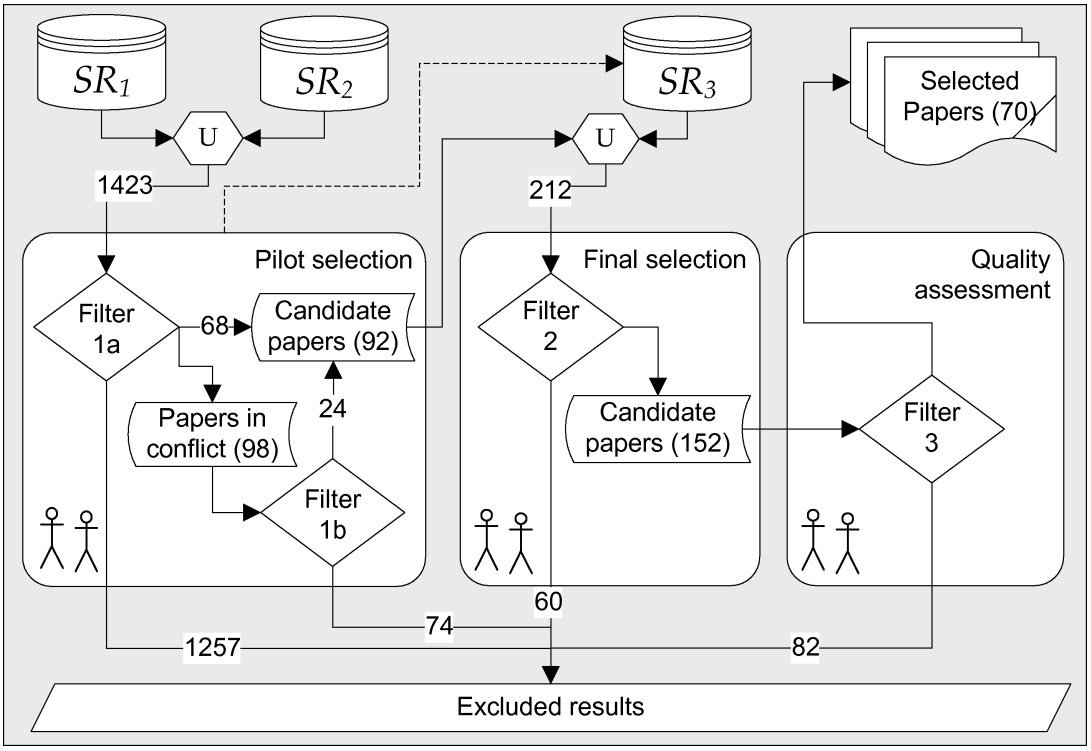

Figure 2.2: *Study selection and quality assessment process.*

Study selection

The study selection is an exclusion process of two phases, i.e. pilot selection and final selection, to evaluate the three sets of search results (see Figure 2.2). 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: 1423 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 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 1257 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 1331. 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 was 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. bioinformatics (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.

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 cannot 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. Table 2.2 lists the selected papers grouped by the venue and the respective venue type. In Tables 2.7, 2.8, and 2.9, 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.

2.3.2 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 extracting two types of data from each of the selected papers. First, the bibliographical references, including the paper's title, authors, venue (journal or conference), and the URL for the digital version. Second, a set of relevant properties per 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.

Table 2.2: Summary of venues and selected articles

Type	Venue	Selected papers
Conference	APAQS: Asia-Pacific Conf. on Quality Software	(Chen et al. 2000)
	ASE: IEEE/ACM Intl. Conf. on Automated Software Engineering	(Vieira and Richardson 2002, Breivold et al. 2008)
	ASPLOS: Intl. Conf. on Architectural support for programming languages and operating systems	(Narayanasamy et al. 2006)
	CAiSE: Intl. Conf. Advanced Information Systems Engineering	(Khan et al. 2008)
	CASCON: Conf. of the Center for Advanced Studies on Collaborative research	(Ronen et al. 2006)
	COMPSAC: Intl. Computer Software and Applications Conf.	(McComb et al. 2002, Moraes et al. 2005)
	CSMR: European Conf. on Software Maintenance and Reengineering	(Xiao and Tzerpos 2005)
	ICAC: Intl. Conf. on Autonomic Computing	(Li, Zhang and Hou 2005)
	ICCS: Intl. Conf. on Conceptual Structures	(Cox et al. 2001)
	ICCSA: Intl. Conf. on Computational Science and Applications	(Mao et al. 2007)
	ICFCA: Intl. Conf. Formal Concept Analysis	(Pfaltz 2006)
	ICPC: Intl. Conf. on Program Comprehension	(Lienhard et al. 2007)
	ICSE: Intl. Conf. on Software Engineering	(Law and Rothermel 2003b, Maule et al. 2008, Vieira et al. 2001, Zimmermann and Nagappan 2008)
	ICSEA: Intl. Conf. on Software Engineering Advances	(Alzamil 2007)
	ICSM: Intl. Conf. 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, Jasz et al. 2008, Korel et al. 2002)
	ISCC: Intl. Conf. on Computers and Communications	(Keller et al. 2000)
	MODELS: Intl. Conf. on Model Driven Engineering Languages and Systems	(Garousi et al. 2006)

Continued on Next Page...

Table 2.2 – Continued

Type	Venue	Selected papers
	OOPSLA: Conf. on Object-oriented programming, systems, languages, and applications	(Sangal et al. 2005)
	OTM: On the Move to Meaningful Internet Systems Confederated Conf.s	(Xiao and Urban 2008)
	PDPTA: Intl. Conf. on Parallel and Distributed Processing Techniques and Applications	(Keller and Kar 2000)
	QSIC: Intl. Conf. on Quality Software	(Liangli et al. 2006)
	SERA: Intl. Conf. on Software Engineering Research, Management and Applications	(Huang and Song 2007, Vasilache and Tanaka 2005)
	TCS: Intl. Conf. on Testing Computer Software	(Ryser and Glinz 2000)
	VLDB: Intl. Conf. on Very large data bases	(Steinle et al. 2006)
	WCRE: Working Conf. on Reverse Engineering	(Callo Arias et al. 2008, Moise and Wong 2005)
Journal	SN: ACM Sigplan Notices	(Chen et al. 2002, Li, Zhou, Wang and Mo 2005)
	IJSEKE: Intl. 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, Gold, Harman and Li 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 Computing	(Bohnet et al. 2009)
	ESEM: Intl. Symposium on Empirical Software Engineering and Measurement	(Cataldo et al. 2008, Nagappan and Ball 2007)
	IM: IFIP/IEEE Intl. Symposium on Integrated Network Management	(Brown et al. 2001)
	ISCIS: Intl. Symposium on Computer and Information Sciences	(Jourdan et al. 2006)
	ISSRE: Intl. Symposium on Software Reliability Engineering	(Law and Rothermel 2003a, Zimmermann and Nagappan 2007)

Continued on Next Page...

Table 2.2 – Continued

Type	Venue	Selected papers
	ISSTA: Intl. symposium on Software testing and analysis	(Xin and Zhang 2007)
	METRICS: Intl. Symposium on Software Metrics	(Leitch and Stroulia 2003)
	NOMS: Network Operations and Management Symposium	(Gao et al. 2004)
	SFM: Intl. School on Formal Methods: Software Architectures	(Stafford et al. 2003)
Workshop	DSOM: IFIP/IEEE Intl. Workshop on Distributed Systems: Operations and Management	(Agarwal et al. 2004, Gupta et al. 2003)
	IWPC: Intl. Workshop on Program Comprehension	(Chen and Rajlich 2000)
	IWPSE: Intl. Workshop on Principles of Software Evolution	(Zhao 2002)
	MSR: Intl. 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: Intl. Workshop on Visualizing Software for Understanding and Analysis	(Holmes and Walker 2007)

2.3.3 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 Figure 2.1). Thus, we focused on the analysis of the extracted data to answers our first three research questions (see Table 2.1). The results are presented in Section 2.4, Section 2.5, and Section 2.6 as summaries and categorizations that practitioners can use as overviews at first and then, if needed, as links or references to investigate details.

2.3.4 Interpretation

In the interpretation phase we aimed at collecting the information to answer our fourth research question (see Table 2.1). The interpretation process started with presenting the summaries for RQ1 (see Section 2.4), RQ2 (see Section 2.5), and RQ3 (see Section 2.6) to the practitioners. We iterated several times to agree on the content and format of the summaries, and to capture the practitioners' perception. We 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 2.7).

2.4 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.

2.4.1 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, Myers, and Constantine (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 et al. (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 et al. have removed the notion of strength from their definition.

Most of the more recent definitions (Mehta et al. 2000, Loyall and Mathisen 1993, Podgurski and Clarke 1990, Stafford and Wolf 1998, Cox et al. 2001) 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 that 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. (Brown et al. 2001). Examples of these dependen-

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

2.4.2 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:

Structural dependencies : Often, when talking about dependencies, what is actually meant are structural (Stafford and Wolf 1998, Allen and Garlan 1997) 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) (Stevens et al. 1974, Podgurski and Clarke 1990, Stafford and Wolf 1998, Allen and Garlan 1997, Myers 1975, Balmas et al. 2005). 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. (Mehta et al. 2000, Stafford and Wolf 1998, Allen and Garlan 1997, Li, Zhou, Wang and Mo 2005). 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 (Gotel and Finkelstein 1994, Watkins and Neal 1994, Egyed 2003). 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.

2.5 Application Areas of Dependency Analysis

In this section we aim to answer our second research question, *Why is dependency analysis needed?* (see Table 2.1). The answer to this question is an overview of a set of activities that dependency analysis solutions, in the literature, claim to support or address. The motivation to build the answer in this way is also observation in practice. 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 neither 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. Second is to identify whether dependency analysis support the activities that practitioners actually perform. Table 2.3 lists the 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, the table 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 the identified application areas and the contribution of dependency analysis to each of them.

Table 2.3: Summary of selected articles per application areas

	Application Areas						
Paper	App. level analysis and management	Arch. description and analysis	Change impact analysis	Program or System understanding	Quality assurance, testing and debug.	Refactoring and modularization	Traceability and feature analysis
Agarwal2004	*						
Alzamil2007						*	
Binkley2005			*				
Bohnet2009					*		
Breivold2008			*				
Brown2001	*						
Cataldo2008						*	
Chen2000				*			
Chen2000a				*			*

Continued on Next Page...

Table 2.3 – Continued

	Application Areas						
Paper	App. level analysis and management	Arch. description and analysis	Change impact analysis	Program or System understanding	Quality assurance, testing and debug.	Refactoring and modularization	Traceability and feature analysis
Cossette2007				*			
Dong2007						*	
Egyed2003							*
Eisenbarth2001							*
Eisenbarth2003							*
Gao2004	*						
Garousi2006					*		
Glorie2009			*				
Gupta2003	*						
Hassan2004			*				
Holmes2007				*			
Huang2007			*				
Ishio2004					*		
Ivkovic2006							*
Jasz2008				*			
Jiang2008				*			
Jourdan2006					*		
Kagdi2007			*				
Keller2000	*						
Keller2000a	*						
Khan2008			*				
Korel2002					*		
Law2003a			*				
Leitch2003						*	
Li2005	*						
Lienhard2007							*
Mao2007			*				
Maule2008			*				
McComb2002	*						
Moise2005				*			
Moraes2005					*		
Nagappan2007					*		
Narayanasamy2006					*		

Continued on Next Page...

Table 2.3 – Continued

	Application Areas						
Paper	App. level analysis and management	Arch. description and analysis	Change impact analysis	Program or System understanding	Quality assurance, testing and debug.	Refactoring and modularization	Traceability and feature analysis
Pfaltz2006				*			
Robillard2008			*				
Ronen2006	*						
Ryser2000					*		
Sangal2005		*					
Stafford2001		*					
Stafford2003		*					
Steinle2006	*						
Vasilache2005							*
Vieira2001		*					
Vieira2002		*					
Xiao2008	*						
Xin2007				*			
Xing2006			*				
Zhang2007					*		
Zhao2001		*					
Zhao2002			*				
Zimmermann2007					*		
Zimmermann2008					*		

2.5.1 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 2.3) 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 auto-

mated distributed management (Keller and Kar 2000), problem determination (Brown et al. 2001, Li, Zhang and Hou 2005, Gao et al. 2004, Gupta et al. 2003, Agarwal et al. 2004), root cause analysis and fault localization (Steinle et al. 2006), and maintaining the correctness of concurrency in multi processes systems (Xiao and Urban 2008).

2.5.2 Architecture Description and Analysis

Due to the increasing size and complexity of software systems, architectural descriptions have 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 document and analyze many aspects of a system's structure and behavior, including dependencies at an architecture 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 2.3). 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).

2.5.3 Change Impact Analysis

Dependency analysis is often applied to assess the impact of changes, i.e. change impact analysis. Change impact analysis is used by practitioners (e.g., architects, designers, developers, and testers) 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 2.3). 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, other solutions support specific activities around change impact analysis such as change prediction (Hassan and Holt 2004, Law and Rothermel 2003b, Kagdi and Maletic 2007), 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 development (Glorie et al. 2009), and requirement change impact on architectural elements (Khan et al. 2008).

2.5.4 Program/System Understanding

It is well known that practitioners spend a considerable amount of time studying artifacts such as source code and documentation as part of the maintenance. This is often necessary to gain a sufficient level of understanding about the system that is developed or maintained. Program and system understanding is another popular area supported by dependency analysis solutions (see Table 2.3). Dependency analysis solution supports the understanding of systems like concurrent software systems (Chen and Rajlich 2000) and multi-languages/poly-lingual systems (Moise and Wong 2005, Cossette and Walker 2007). The main activities around program understanding supported by dependency analysis include the identification of particular types of dynamic dependencies (Xin and Zhang 2007, Jasz et al. 2008, Pfaltz 2006), program slicing analysis (Jiang, Gold, Harman and Li 2008), and reuse analysis (Holmes and Walker 2007).

2.5.5 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 supports quality assurance through planning and implementation activities (see Table 2.3). 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 2008, Zimmermann and Nagappan 2007). Forecasting the load level of system components is supported by analyzing behavioral dependencies on model designs. This type of analysis 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 at hand, 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 testing activities supported by dependency analysis are 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 guide the insertion of 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 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 program-

ming error or irrespective use of the language specification, e.g., using a variable before it has been defined. Dependency analysis supports debugging by 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 et al. 2006), and even debugging of aspect-oriented software (Ishio et al. 2004).

2.5.6 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). 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 interdependencies 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 might have 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).

2.5.7 Traceability and Feature Analysis

Software development artifacts such as model descriptions, specifications, and source code are highly interrelated. Changes in one artifact might cause changes in another producing 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 2.3). 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 out-dated, which causes that understanding the system becomes harder every time a feature is changed. Depen-

dency 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. 2003, Eisenbarth et al. 2001, Lienhard et al. 2007).

2.6 Existing Dependency Analysis Solutions

In this section, we present the answer to our third research question (see Table 2.1). The answer describes existing dependency analysis solutions groped 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 2.5). The sources of information used among existing dependency solutions can be classified as: 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 2.2.2). 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.

2.6.1 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). Table 2.4 shows that among solutions that analyze source code data one can distinguish three groups based on the analysis approach: static, dynamic, and change history analysis. The table maps the type of analysis approach (or combination of approaches) to the application area and the reference of 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) (Podgurski and Clarke 1990, Ferrante et al. 1987). 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).

Table 2.4: *Dependency analysis solutions using source code-based information*

Approach	Application Areas	References
Static Analysis	Arch. description and 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 or system understanding	(Chen and Rajlich 2000, Cossette and Walker 2007, Holmes and Walker 2007, Jiang, Gold, Harman and Li 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, Law and Rothermel 2003b, Tallam and Gupta 2007)
	Program or system understanding	(Jasz 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)
Static + Dynamic Analysis	Application level analysis	(Ronen et al. 2006)
	Quality assurance	(Zimmermann and Nagappan 2008, Zimmermann and Nagappan 2007)
	Feature analysis	(Eisenbarth et al. 2001, Eisenbarth et al. 2003)
Static + Dynamic + Change Hist. Analysis	Refactoring and modularization, change impact analysis, and debug.	(Bohnet et al. 2009)

Static analysis

Dependency analysis solutions based on static analysis are extensions of the PDG (Chen and Rajlich 2000, Ishio et al. 2004, Leitch and Stroulia 2003, Chen et al. 2000, Zhao 2002)

and the Dependency (or Design) Structure Matrix (DSM) (Sangal et al. 2005, Breivold et al. 2008). A variety of static analysis techniques are used by dependency solutions to discover and present 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, Gold, Harman and Li 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 2.4).

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). 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++ (Lienhard et al. 2007, Egyed 2003).

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 2003b, Law and Rothermel 2003a), object flow analysis (Lienhard et al. 2007), redundant coupling detection (Alzamil 2007), online detection (Xin and Zhang 2007), execute after/before analysis (Jasz 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 2.4 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).

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 2.4 illustrates the application areas supported by these various techniques.

Combining information sources

Some dependency analysis solutions propose combinations of analysis approaches to increase completeness and precision of dependency information. 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 2.4). The techniques used by solutions that combine static and dynamic analysis include concept analysis (Eisenbarth et al. 2003, Eisenbarth et al. 2001), pattern languages (Ronen et al. 2006), and network analysis (Zimmermann and Nagappan 2008, Zimmermann and Nagappan 2007). 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.

2.6.2 Descriptions and Model-based solutions

System documentation includes diagrammatic and semi-formal descriptions about the structure and behavior of a software system at a high level of abstraction. These diagrammatic descriptions include representations such as UML diagrams and sketches with blocks and arrows. Other semi-formal descriptions include descriptions using Architectural Description Languages (ADLs) and Interface Description Languages (IDLs), which are initiatives of the research community on software architecture and component-based platform to describe systems at an architectural level.

Table 2.5 summarizes 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.

Diagrammatic Descriptions

Dependency analysis solutions 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 (Mao et al. 2007, Xing and Stroulia 2006, Khan et al. 2008), chart diagrams (Garousi et al. 2006, Moraes et al. 2005, Ryser

Table 2.5: *Dependency analysis solutions using description and model-based information*

Approach	Application Areas	References
Analysis of Diagrammatic Descriptions	App. level analysis and management	(McComb et al. 2002)
	Architecture description and 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 et al. 2005)
	Traceability	(Ivkovic and Kontogiannis 2006, Vasilache and Tanaka 2005)
Analysis of Semi-formal Descriptions	Architecture description and 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)

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 (Mao et al. 2007, Xing and Stroulia 2006, Khan et al. 2008).

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 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.

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 ADL descriptions. 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 (Korel et al. 2002, Jourdan et al. 2006) 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 an original ESFM model and a modified model, which help to identify the modified elements and support the reduction of regression testing activities.

2.6.3 Run-time monitored and Configuration-based solutions

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

Table 2.6: *Dependency analysis solutions using run-time and configuration information*

Approach	**Application Areas**	**References**
Analysis of Monitored data	Application level analysis and management	(Agarwal et al. 2004, Li, Zhang and Hou 2005, 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 et al. 2006)
Analysis of Configuration Repositories	Application level analysis and management	(Keller et al. 2000, Keller and Kar 2000)

Table 2.6 summarizes the set of dependency analysis solutions that use run-time monitored data and configuration repositories. These solutions share a number of characteristic. First, most of these solutions are dedicated to support application level analysis and management activities (see Section 2.5.1). Second, these solutions see major elements of a running system (subsystems, applications, services, data repositories etc.) as black boxes. Third, these solutions focus on the identification and analysis of structural and behavioral dependencies between major system elements.

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 offline using data mining algorithms to construct dependency models (Steinle et al. 2006, Xiao and Urban 2008, Callo Arias et al. 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, Li, Zhang and Hou 2005, Brown et al. 2001, Gupta et al. 2003). Other solutions use hardware-based monitoring mechanisms to capture shared memory dependencies for supporting debugging activities (Narayanasamy et al. 2006).

Configuration repositories

Configuration repositories provide 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.

2.7 Applicability of Dependency Analysis

In this section we summarize the assessment of dependency analysis solutions that we conducted in the interpretation phase (see Section 2.3.4). The assessment is composed of two parts: the applicability of definitions related to dependencies in practice (see Section 2.7.1), and the applicability of dependency analysis solutions in practice (see Sections 2.7.2, 2.7.3, and 2.7.4). For each type of dependency analysis solutions we

summarize:

- The value, the practitioners' concerns, and opportunities for improvement that we identified for each solution.
- The quality assessment of the selected papers selected in the quality assessment phase (see Section 2.3.1). For the quality assessment, we take into account two aspects:
  - *Characteristic of the selected paper*. We distinguish the type of system used for the validation of the proposed solution, see column 'Case' in Tables 2.7, 2.8, and 2.9. The identified values are 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.
  - *Practitioners' perception*. According to the practitioners perception, we distinguish whether the solution is easy to use and ready to use in practice, see columns 'Easy to Use', and 'Ready to Use' in Tables 2.7, 2.8, and 2.9. 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 Use, there are three possible values. Y (yes) means that tool-support and a well defined process are readily available for the proposed solution, e.g. a tool can be downloaded from a public website. P (partly) means that we can implement the tool-support and the application process of the proposed solution without considerable effort. N (no) means that there are major concern about identifying and implementing the tool-support and application process for the propose solution.

2.7.1 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 due to two major aspects:

Generalization : The definitions only concern specific types of dependencies and thus have a very narrow scope. 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.

Context : Definitions do not take into account the context of the system or the organization. For practitioners 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 aspects, we came up with a definition, based on the literature, our observations, and discussions with practitioners at Philips Healthcare MRI. 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.

2.7.2 Source code-based solutions

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. When dependencies are related to tangible and explicit constructs, that one can point out directly in the source code, analyzing and resolving problems around them is a relatively straightforward activity for practitioners. This makes that source code-based solutions are very pleasant for practitioners (e.g., programmers and designers) and partly explains the appeal these solutions have on them.

In practice, however, there are some concerns about the scalability of solutions using static analysis:

Data volume : For large and complex implementations, static analysis solutions can generate enormous volume of data (in the form of dependency graphs). A considerable effort is necessary to extract or identify actual dependencies from this large volume of data. Part of this problem can be addressed raising the granularity used by most static analysis methods, which does not align with the high-level perspective of practitioners like software architects and managers.

Heterogeneity : 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 uses 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 of static analysis solutions. For example, heterogeneity is addressed in (Kontogiannis et al. 2006) for large multi-language software systems. However, mechanisms and extensions of static cross-language analysis are still expensive and miss dependencies due to dynamic behavior (Moise and Wong 2005, Cossette and Walker 2007). Thus, static analysis solutions do need to be complemented at least with dynamic analysis solutions. In practice, we observed that such combination is needed 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.

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 the system at hand has a heterogeneous implementation.

However, when working with dynamic analysis solutions in practice, practitioners concern about two main aspects:

Overhead : Practitioners have to manage 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 large and heterogeneous implementation, including components with partial or no source code available, produce 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.

Data volume : Practitioners 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 summarization and visualization techniques (Safyallah and Sartipi 2006, Cornelissen et al. 2007), but also choosing higher levels of abstraction.

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 2.7.1 we described that interesting dependencies are those that create 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, regardless of how the dependency is 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. If the source code structure of a system change, relationships identified using the historical data of the previous structure may not longer exist.

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 them 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 2.4. 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.

Table 2.7: *Assessment of source code-based solutions*

Reference	Definition	App. 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
(Jasz et al. 2008)	P	Y	OS	N	P
(Jiang, Gold, Harman and Li 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

2.7.3 Model-based solutions

Diagrammatic Descriptions

Practitioners i.e. software architects and designers use diagrammatic descriptions such as design models and architectural views to facilitate the communication between them during development projects. We observed that conducting dependency analysis using design models and documents helps to identify dependencies between high-level components. For instance, when phasing-out a legacy component, architects and designers analyze design documentation to plan the phasing-out process. They try to identify all 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 are and how to remove them. Later on, during the implementation 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 a important factor:

Up-to-date descriptions : 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.

Semi-formal Descriptions

ADLs (Stafford and Wolf 2001, Zhao 2001) and EFSM (Korel et al. 2002, Jourdan et al. 2006) have emerged as potential solutions for formally describing system architectures. 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 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.

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 2.7.2.

Applicability in practice

In brief, solutions using diagrammatic and semi-formal descriptions are more appealing for practitioners following architecture-driven approaches. These 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.

2.7.4 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, implementation technology borders do not limit these solutions. 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 only monitored scenarios and the components whose configuration was analyzed.

The way that these solutions analyze major system components as black boxes needs to be adjusted to contribute to 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).

Table 2.8: *Assessment of description and model-based solutions*

Reference	Definition	App. 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

Combining information sources

Combining sources of information is an approach that source code-based solutions 2.6.1 use as a way for coping with incompleteness. We observed that combining different sorts of run-time monitored data like logging and process activity (Callo Arias et al. 2008), helps to link system run-time components (seen as black boxes) to source code related artifacts, e.g., binaries and libraries.

Applicability in practice

In brief, we found that run-time monitored and configuration-base solutions 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 2.6.1).

Table 2.9: *Assessment of run-time and configuration-based solutions*

Reference	Definition	App. 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, Zhang and Hou 2005)	N	P	Toy	N	N
(Narayanasamy et al. 2006)	Y	Y	Other	N	N
(Steinle et al. 2006)	P	Y	Industrial	N	N
(Xiao and Urban 2008)	Y	Y	Other	N	N

2.8 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 generalization of the results for other domains).

2.8.1 Internal validity

- **About the study search:** Below are some of the aspects that can threat the validity of our 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 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. (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.

An alternative for the study search is a manual search on specific venues. 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 2.2, 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 2.2 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 Wusteman 2007). This search engine allowed us to search across different digital libraries (e.g., IEEEXplore and ACM Portal) and cove various venues (see Table 2.2). 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, 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 2004a)). The second is using search engines of dedicated databases of research literature. For example, Scopus[2] is an option that could support our search queries and perhaps provide less false positives. However, engines of this

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

type are not yet known and accessible to most practitioners.

- **About the study selection:** Through the study selection process, we noticed the risk of bias due to our research interest 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 2.3.1), yet this process involved two reviewers. An alternative could be the inclusion of a third-party reviewer with a particular or broad research interest about dependency analysis. Due to the practical nature of our project, we needed to deliver results to the practitioners in a timely manner. Therefore, we decided not to extend the study search during the process. 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 additional relevant studies, which could influence the practitioners' perception and the results of the review.

- **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.

- **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: during the synthesis process merging the extracted data to construct the summaries, 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.

2.8.2 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. Consequently, a potential threat to validity is that the information in the overview is specific to our research context and cannot be generalized to other organizations or domains. We consider that the MRI scanner 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 to 30 microprocessors, with software controlling aspects such as engine ignition, pollution control, security, 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.

2.9 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 describe 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 scanner.

Researchers and practitioners can use the overview to learn about the state-of-the-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 2.7. 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.

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 Philips MRI scanner, 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 not only describe practical issues, but also expose research opportunities that can have a strong impact on the way that software organizations develop large and complex software-intensive systems.