Towards A Process-Oriented Software Architecture Reconstruction Taxonomy

Damien Pollet Stéphane Ducasse Loïc Poyet Ilham Alloui Sorana Cîmpan Hervé Verjus

LISTIC, Université de Savoie, France

E-mail: ⟨damien.pollet,stephane.ducasse⟩@univ-savoie.fr

Abstract

To maintain and understand large applications, it is crucial to know their architecture. The first problem is that unlike classes and packages, architecture is not explicitly represented in the code. The second problem is that successful applications evolve over time, so their architecture inevitably drifts. Reconstructing the architecture and checking whether it is still valid is therefore an important aid. While there is a plethora of approaches and techniques supporting architecture reconstruction, there is no comprehensive state of the art and it is often difficult to compare the approaches. This article presents a state of the art on software architecture reconstruction approaches.

1. Introduction

Software architecture acts as a shared mental model of a system expressed at a high-level of abstraction [50]. By leaving details aside, this model plays a key role as a bridge between requirements and implementation. It allows you to reason architecturally about a software application during the various steps of the software life cycle. According to Garlan, software architecture plays an important role in at least six aspects of software development: understanding, reuse, construction, evolution, analysis and management [34].

Software architecture is thus crucial for software development, but architectures are not explicitly represented in the code as classes and packages. Another problem is that successful software applications are doomed to continually evolve and grow [71]; and as a software application evolves and grows, so does its architecture. The conceptual architecture often becomes inaccurate with respect to the implemented architecture; this results in architectural erosion [83, 98], drift [98], mismatch [35], or chasm [106].

Several approaches and techniques have been proposed in the literature to support Software architecture reconstruction (SAR). Mendonça *et al.* presented a first raw classification of SAR environments based on a few typical scenarios [85]. O'Brien *et al.* surveyed SAR practice needs and approaches [95]. Still, there is no comprehensive state of the art and it is often difficult to compare the approaches. This article presents a state of the art of software architecture reconstruction approaches. It takes the perspective of a reverse engineer who would like to reconstruct the architecture of an existing application and would like to know which tools or approaches to consider. We structure the study around the processes, the inputs, the techniques and the outputs of SAR approaches and we propose a taxonomy for SAR in this context.

About selecting the approaches. In this paper, we select works in two steps. First, in addition to works that extract architectural information, we consider approaches that visualize programs since they are often the basis for abstracting and extracting architectural views, but we limit ourselves to the program visualization approaches that support the overall extraction process. For the sake of space, we exclude approaches that do not specifically extract architecture but related artefacts such as design patterns, features or roles.

In a second step, we support the comparison of the approaches with a table for each axis that structures this survey. We only list in the tables works that are the most concerned about architectural extraction. For the sake of space again, we consider only two categories of works: the important ones *i.e.*, those which were influential or were precursors, and the original works taking a specific approach to the general problem. This latter category is interesting because it opens the survey space.

Section 2 describes the criteria that we adopted in our taxonomy. Sections 3 to 6 then cover each of those criteria before concluding.

2. SAR taxonomy axes

We propose a deeper classification (Fig. 1) based on the life time of SAR approaches: intended goals, followed processes, required inputs, used techniques and expected outputs. Our taxonomy treats a larger number of approaches than the previous attempts at classifying the field.

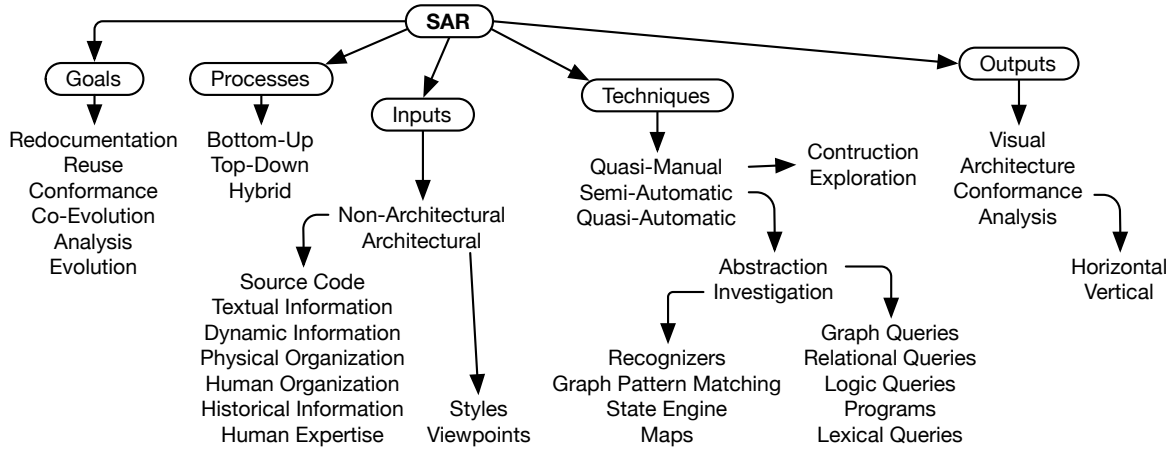

Figure 1. A process-oriented taxonomy for SAR.

Goals. SAR is considered by the community as a proactive approach to answer stakeholder's business goals [23, 118]. The reconstructed architecture is the basis for redocumentation, reuse investigation and migration to product lines, or co-evolution of implementation and architecture. Some approaches do not extract the architecture itself but related and orthogonal artifacts that provide valuable additional information to engineers such as design patterns, roles or features. For sake of space, as previously said, we do not expand further this axis.

Processes. We distinguish three kinds of SAR processes based on their flow to identify an architecture: bottom-up, top-down or hybrid.

Inputs. Most SAR approaches are based on source code information and human expertise. However, some of them exploit other architectural or non-architectural information sources such as dynamic information or historical information. It is worth noting that not all approaches use architectural styles and viewpoints even though those are the paramount of software architecture.

Techniques. The research community has explored various architecture reconstruction techniques that we classify according to their level of automation.

Outputs. While all SAR approaches intend to provide architectural views, some of them produce other valuable outputs such as information about the conformance of architecture and implementation.

3. SAR processes

SAR follows either a bottom-up, a top-down or an hybrid opportunistic process.

3.1. Bottom-up processes

Bottom-up processes start with low-level knowledge to recover architecture. In most cases, from source code models, they progressively raise the abstraction level until a high-level understanding of the application is reached [9, 120] (see Fig. 2).

Also called architecture *recovery* processes, bottom-up processes are closely related to the well-known *extract-abstract-present* cycle described by Tilley *et al.* [129]. Source code analyses populate a repository, which is queried to yield abstract system representations, which are then presented in a suitable interactive form to reverse engineers.

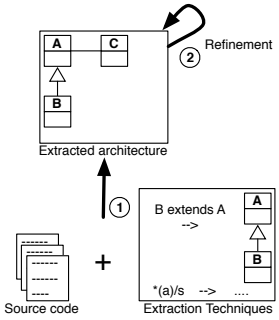

Figure 2. A bottom-up process: from the source code, views are (1) extracted and (2) refined.

Examples. The Dali tool by Kazman *et al.* [56, 57] supports a typical example of a bottom-up process: (1) Heterogeneous low-level knowledge is extracted from the software implementation, fusioned and stored in a relational database; (2) Using the Rigi visualization tool [91, 128], a reverse engineer visualizes and manually abstracts this information; (3) A reverse engineer specifies patterns by selecting source model entities with SQL queries and abstracting them with Perl expressions. Based on Dali, Guo *et al.* proposed ARM [40] which focuses on design patterns conformance.

In Intensive, Mens *et al.* use intension logic to group

related source-code entities [87, 145]. Reverse engineers incrementally define views and their relations with intensions specified as Smalltalk or logic queries. Intensive classifies the views and displays consistencies and inconsistencies with the code and between architectural views. Intensive visualizes its results with CodeCrawler [70].

Lungu *et al.* built both a method and the Softwarenaut tool [77] to interactively explore packages. They enhance the exploration process in the package architectural structure by guiding the reverse engineer towards the relevant packages. They characterize packages based on their relations and on their internal structure. A set of packages are highlighted and associated to exploration operations that indicate to the reverse engineer the actions to perform to get a better understanding of the software architecture.

Other bottom-up approaches include ArchView [99], Revealer [100, 101], ARES [26], ARMIN [58] and Gupro [24]. We classify the works around PBS/SBS in this category, but since they consider conceptual architectures to steer the process, we could as well have classified them with the hybrid processes [8, 31, 49, 113].

3.2. Top-down processes

Top-down processes start with high-level knowledge such as requirements or architectural styles and aim to discover architecture by formulating conceptual hypotheses and by matching them to the source code [12, 92, 120] (see Figure 3). The term architecture *discovery* often describes this process.

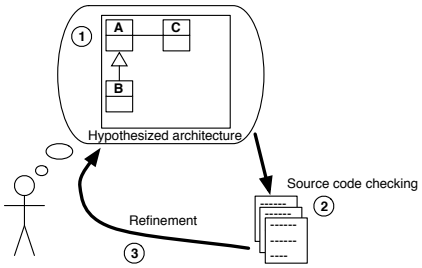

Figure 3. A top-down process: (1) an hypothesized architecture is defined, (2) the architecture is checked against the source code, (3) the architecture is refined.

Examples. The Reflexion Model of Murphy *et al.* is a typical example of a top-down process [92, 93]. First, the reverse engineer defines his high-level hypothesized conceptual view of the application. Second, he specifies how this view maps to the source code concrete view. Finally, RMTool confronts both conceptual and concrete views to compute a reflexion model that highlights *convergences*, *divergences* and *absences*. The reverse engineer iteratively computes and interprets reflexion models until satisfied. In a reflexion model, a convergence locates an element that is present in both views, a divergence an element that is only in the concrete view, and an absence an element that is only in the conceptual view. The reflexion model offers a better support to express the conceptual architecture and the results of the process than the approach developed in SoFi [12]. The reflexion model influenced other works [13, 44, 61, 105, 132].

3.3. Hybrid processes

Hybrid processes combine bottom-up with top-down processes [120]. On one hand, low-level knowledge is *abstracted* using various techniques. On the other hand, high-level knowledge is *refined* and confronted against the previously extracted views. By reconciling the conceptual and concrete architectures, hybrid processes are frequently used to stop architectural erosion [83, 98]. Hybrid approaches often use hypothesis recognizers that provide bottom-up reverse engineering strategies to support top-down exploration of architectural hypotheses [97].

Examples. Sartipi implements a pattern-based SAR approach in Alborz [110]. The architecture reconstruction has two phases. During the first bottom-up phase, Alborz parses the source code, presents it as a graph, then divides that graph in cohesive regions using data mining techniques. The resulting model is at a higher abstraction level than the code. During the second top-down phase, the reverse engineer iteratively specifies his hypothesized views of the architecture in terms of patterns. These patterns are approximately mapped with graph regions from the previous phase using graph matching and clustering techniques. Finally, the reverse engineer decides to proceed or not to a new iteration based on the partially reconstructed architecture and evaluation information that Alborz provides.

Christl *et al.* present an evolution of the Reflexion Model [13]. They enhance it with automated clustering to facilitate the mapping phase. As in the Reflexion Model, the reverse engineer defines his hypothesized view of the architecture in a top-down process. However, instead of manually mapping hypothetic entities with concrete ones, the new method introduces clustering analysis to partially automate this step. The clustering algorithm groups concrete entities that are not mapped yet with similar concrete entities already mapped to hypothesized entities.

To assess the creation of product lines, Stoermer *et al.* introduce the MAP method [117]. MAP combines (1) a bottom-up process to recover the concrete architectures of existing products; (2) a top-down process to map architectural styles onto recovered architectural views; (3) an approach to analyze commonalities and variabilities among recovered architectures. They stress the ability of architectural styles to act as the structural glue of the components, and to highlight architecture strengths and weaknesses.

Other hybrid processes include Focus [18, 84], Nimeta [106], ManSART [4, 43], ART [32], X-Ray [86], ARM [40] and DiscoTect [146]. In ManSART, a top-down recognition engine maps a style-compliant conceptual view with a system overview defined in a bottom-up way using a visualization tool [4, 43].

Table 1. SAR process overview

Alborz [110]			hybrid
ArchView [99]	bottom-up		
ArchVis [45]	bottom-up		
ARES [26]	bottom-up		
ARM [40]			hybrid
ARMIN [58]	bottom-up		
ART [32]			hybrid
Bauhaus [13, 25, 62]			hybrid
Bunch [79, 90]	bottom-up		
Cacophony [28]			hybrid
Dali [56, 57]	bottom-up		
DiscoTect [146]			hybrid
Focus [18, 84]			hybrid
Gupro [24]	bottom-up		
Intensive [87, 145]	bottom-up		
ManSART [4, 43]			hybrid
MAP [117]			hybrid
PBS/SBS [8, 31, 49, 113]			hybrid
PuLSE/SAVE [61, 103]		top-down	
QADSAR [118, 119]			hybrid
Revealer [100, 101]	bottom-up		
RMTool [92, 93]		top-down	
SARTool [30, 64]	bottom-up		
SAVE [89, 94]		top-down	
Softwarenaut [77]	bottom-up		
... with Hapax [67, 76, 77]	bottom-up		
Symphony,Nimeta [106, 135]			hybrid
URCA [6]	bottom-up		
W4 [44]		top-down	
X-Ray [86]			hybrid
— [7]			hybrid
— [51]			hybrid
— [75]	bottom-up		
— [97]			hybrid
— [132]			hybrid

4. SAR Inputs

Most often, SAR works from source code representations, but it also considers other kinds of information, such as dynamic information extracted from a system execution, or historical data held by version control system repositories. A few approaches work from architectural elements such as styles or viewpoints. There is no clear trend because SAR approaches are fed with heterogeneous information of various abstraction levels. In this section, we present first the non-architectural inputs, then the architectural inputs.

4.1. Non-architectural inputs

Source Code Constructs. The source code is an omnipresent trustworthy source of information that most approaches consider. Some of the approaches directly query the source code using regular expressions like in RMTool [92, 93] or [100, 101]. However, most of them do not work from the source code text but represent it using metamodels. These metamodels cope with the paradigm of the analyzed software. For instance, the language independent metamodel FAMIX is used to reverse engineer object-oriented applications [17]; its concepts include classes, methods, calls or accesses. FAMIX is used in ArchView [99], Softwarenaut [77] and Nimeta [106]. Other metamodels such as the Dagstuhl Middle Metamodel [72] or GXL [48] have been proposed with the same intent of abstracting the source code.

Symbolic Textual Information. Some approaches work from the symbolic information available in the comments [100, 101] or in the method names [66].

Dynamic Information. Static information is often insufficient for SAR since it only provides a limited insight into the runtime nature of the analyzed software; to understand behavioral system properties, dynamic information is more relevant [68]. Some SAR approaches use dynamic information alone [138, 146] while others mix static and dynamic knowledge [51, 73, 99, 107, 137]. A lot of approaches using dynamic information extract design views rather than architecture [41, 42, 104, 125, 126]. Huang *et al.* consider runtime events such as method calls, CPU utilization or network bandwidth consumption because it may inform reverse engineers about system security properties or system performance aspects [51]. DiscoTect uses dynamic information too [146]. Li *et al.* use run-time process information to derive architectural views [73]. Some works focus on dynamic software information visualization [21, 54, 126]. To get a more precise analysis of these works, we refer the reader to the survey of Hamou-Lhadj *et al.* [42]. Dynamic information is also used to identify features [25, 38, 109], design patterns [46, 139], or collaborations and roles [105, 142].

Physical Organization. The physical organization of applications in terms of files and folders often reveals architectural information. ManSART [4, 43] and Softwarenaut [77] work from the structural organization of physical elements such as files, folders, or packages. Some approaches map packages or classes to components and use the hierarchical nature of the physical organization as architectural input [69, 102, 143].

Human Organization. According to Conway [15]: *"Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations"*. It is then important to consider the influence of the human organization on the extracted architectures or views. Inspired by Conway's thesis, Bowman *et al.* use the developer's organization to form an ownership

architecture that helps stakeholders reconstruct the software architecture [7].

Historical Information. Historical information is rarely used in SAR. Wuyts worked on the co-evolution between code and design [145]. ArchView is a recent approach that exploits source control system data and bug reports to analyze the evolution of recovered architectural views [99]. Mens *et al.* analyse the evolution of extracted software views with Intensive [87, 145]. To assist a reverse engineer in understanding dependency gaps in a reflexion model [92, 93], Hassan *et al.* annotate entity dependencies with sticky notes that record dependency evolution and rationale with information extracted from version control systems [44].

Human Expertise. Although one cannot entirely trust human knowledge, it is very helpful when it is available. At high abstraction levels, SAR is iterative and requires human knowledge to guide it and to validate its results. To specify a conceptual architecture [44, 84, 92], reverse engineers have to study system requirements, read available documentation, interview stakeholders, recover design rationale, investigate hypotheses and analyze the business domain. Human expertise is also required when specifying viewpoints, selecting architectural styles (Section 4.2), or investigating orthogonal artifacts. While SAR processes involve strategy and knowledge of the domain and the application itself, only a few approaches take human expertise explicitly into account. Ivkovic *et al.* [53] propose to systematically update a knowledge base that would become a helpful collection of domain-specific architectural artifacts.

4.2. Architectural inputs

Architectural styles and viewpoints are the paramount of software architecture, therefore we analyzed whether SAR approaches consider them as input to steer the extraction process.

Styles. Architectural styles such as pipes and filters, layered system, data flow and blackboard are popular because like design patterns, they represent recurrent architectural situations [11, 59]. They are valuable, expressive, and accepted abstractions for SAR and more generally for software understanding.

Recognizing them is however a challenge because they span several architectural elements and can be implemented in various ways [100, 101]. The question that turns up is whether SAR helps reverse engineers specify and extract architectural styles.

Examples. In Focus, Ding *et al.* use architectural styles to infer a conceptual architecture that will be mapped to a concrete architecture extracted from the source code [18, 84].

Closely related to this work, Medvidovic *et al.* introduce an approach to stop architectural erosion. In a top-down process, requirements serve as high-level knowledge to discover the conceptual architecture [83]. In a bottom-up process, system implementation serves as low level knowledge to recover the concrete architecture. Both the conceptual and the concrete architectures are incrementally built. The reverse engineer reconciles the two architectures, based on architectural styles. Their approach considers architectural styles as key design idioms since they capture a large number of design decisions, their rationale, effective compositions of architectural elements, and system qualities that will likely result from using the style.

DiscoTect reconstructs style-compliant architectures [146]. Using a state machine, DiscoTect incrementally recognizes interleaved patterns in filtered execution traces of the application. The state machine represents an architectural style; by refining it, the reverse engineer defines which hypothesized architectural style the tool should look for [123].

ManSART [4, 43], ART [32] and MAP [117] are other SAR approaches taking architectural styles into account.

Viewpoints. The system architecture acts as a mental model shared among stakeholders [50]. Since the stakeholders' interests are diverse, viewpoints are important aspects that SAR may consider [52, 114]. Viewpoint catalogues were built to address this issue: the $4 + 1$ viewpoints of Kruchten [65]; the four viewpoints of Soni *et al.* [47, 116], the build-time viewpoint introduced by Tu *et al.* [134] or the implicit viewpoints inherent to the UML standard. Most SAR approaches reconstruct architectural views according either to a single viewpoint or a few preselected viewpoints. Smolander *et al.* highlight that viewpoints cannot be standardized but should be selected or defined according to the environment and to the situation [114]. O'Brien *et al.* present the View-Set Scenario pattern that helps determine which architectural views sufficiently describe the system and cover the stakeholders' needs [95].

Examples. The Symphony approach of van Deursen *et al.* aims at reconstructing software architecture using appropriate viewpoints [135]. Viewpoints are selected from a catalogue or are defined if they do not exist, and they evolve throughout the process. They constrain SAR to provide architectural views matching the stakeholders' expectations, and ideally immediately usable. The authors show how to define viewpoints step by step, and applied their approach on four case studies with different stakeholders' goals. They provide architectural views to reverse engineers following the viewpoints those reverse engineers typically use during design phases. Based on Symphony, Riva proposed the view-based SAR approach Nimeta [106].

Favre outlines Cacophony, a generic SAR metamodel-driven approach [28]. Like Symphony, Cacophony recognizes the need to identify the viewpoints that are relevant to the stakeholders' concerns and that SAR must consider. Contrary to Symphony, Cacophony states that metamodels are keys for representing viewpoints.

The QADSAR approach both reconstructs the architec-

ture of a system and drives quality attribute analyses on it [118, 119]. To identify the relevant architectural viewpoints, reverse engineers formulate scenarios that highlight interesting quality attributes of the system. ARES [26] and SARTool [30, 64] also take viewpoints into account.

4.3. Mixing inputs

Most approaches work from a limited source of information, even if multiple inputs are necessary to generate rich and different architectural views. Kazman *et al.* [55] advocate the fusion of multiple source of inputs to produce richer architectural views: for example, they produce interprocess communication and file access views. Lange *et al.* [68] mix dynamic and static views to extract design patterns.

ArchVis [45] works from source code, file structures and dynamic information such as network log or message sends.

Knodel *et al.* [60] discuss the combination of different information sources such as documents, source code and historical data. However it is not clear whether the approach is used in practice. Multiple inputs must be organized and Ivkovitch proposes a systematic way to organize application domain knowledge into a unified structure [53].

5. SAR Techniques

There is a variety of formalisms used to express, query and exchange data representing applications [36, 107]. A couple of exchange formats exist from simple textual tuples in RSF [141] or in TA [8, 31, 49, 113], to XML in GXL [48] and in [24, 106], or to CDIF in FAMIX [17]. The format may limit the merging or manipulation of the information [22]. An important property of an exchange format is that it can be easily generated and used with simple tools [19].

SAR techniques are often correlated with the data they operate on and the formalisms used for their representation and manipulation: for example, Mens *et al.* express logic queries on facts [87, 145] while Ebert *et al.* perform queries on graphs [24]. Thus, instead of using data formalisms as a criterion, we classify techniques into three automation levels:

Quasi-manual. the reverse engineer manually identifies architectural elements using a tool to assist him in understanding his findings;

Semi-automatic. the reverse engineer manually instructs the tool how to automatically discover refinements or recover abstractions;

Quasi-automatic. the tool has the control and the reverse engineer steers the iterative recovery process.

Of course, the boundaries in the classification are not clear-cut. Moreover, reverse engineers often use visualization tools to understand the results of their analyses, but a comparison of the visualization tools is beyond the scope of this article. Table 3 synthesizes the classification of SAR techniques.

Table 2. SAR input overview

	src	text	dyn	phys	org	hist	exp	style	viewp
Alborz [110]	src		dyn				exp		
ArchView [99]	src		dyn			hist	exp		
ArchVis [45]	src	text	dyn	phys				style	viewp
ARES [26]	src						exp		
ARM [40]	src						exp		
ARMIN [58]	src						exp		
ART [32]	src						exp	style	
Bauhaus [13, 25, 62]	src		dyn				exp		
Bunch [79, 90]	src						exp		
Cacophony [28]							exp		viewp
Dali [56, 57]	src						exp		
DiscoTect [146]	src		dyn				exp	style	
Focus [18, 84]	src						exp	style	
Gupro [24]	src						exp		
Intensive [87, 145]	src						exp		
ManSART [4, 43]	src			phys			exp	style	
MAP [117]	src						exp	style	
PBS/SBS [8, 31, 49, 113]	src			phys			exp		
PuLSE/SAVE [61, 103]	src						exp		
QADSAR [118, 119]	src						exp		viewp
Revealer [100, 101]	src	text					exp		
RMTool [92, 93]	src						exp		
SARTool [30, 64]	src						exp		viewp
SAVE [89, 94]	src						exp		
Softwarenaut [77]				phys			exp		
... with Hapax [67, 76, 77]	src	text		phys			exp		
Symphony,Nimeta [106, 135]			dyn				exp		viewp
URCA [6]	src		dyn				exp		
W4 [44]	src					hist	exp		
X-Ray [86]	src						exp		
— [7]	src				org	hist	exp		
— [51]	src		dyn					style	
— [75]	src						exp		
— [97]	src		dyn				exp	style	
— [132]	src						exp		

src source code text textual information dyn dynamic information phys physical organization org human organization hist historical information exp human expertise style styles viewp viewpoints

5.1. Quasi-manual techniques

SAR is a reverse engineering activity which faces scalability issues in manipulating knowledge. In response to this problem, researchers have proposed slightly assisted techniques; we group those into two categories: construction-based techniques and exploration-based techniques.

Construction-based techniques. These techniques reconstruct the software architecture by manually abstracting low-level knowledge, thanks to interactive and expressive visualization tools: Rigi [91, 128], CodeCrawler [70], Shrimp/Creole [121, 122], Verso [69], 3D [81] or GraphViz [33].

Exploration-based techniques. These techniques give reverse engineers an architectural view of the system by guiding them through the highest-level artifacts of the imple-

mentation, like in Softwarenaut [77]. The architectural view is then closely related to the developer's view. Instead of providing guidance, the SAB browser [27] allows reverse engineers to assign architectural layers to classes and then to navigate the resulting architectural views. ArchView[1] [29] visualizes simple architectural elements and their relationships in 3D.

5.2. Semi-automatic techniques

Semi-automatic techniques automate repetitive aspects of SAR. The reverse engineer steers the iterative refinement or abstraction, leading to the identification of architectural elements. As in the quasi-manual techniques, we distinguish two categories: abstraction-based techniques and investigation-based ones.

Abstraction-based techniques. These techniques aim at mapping low-level concepts with high-level ones. Reverse engineers specify reusable abstraction rules and execute them automatically; we identified five approaches:

Graph queries. Gupro queries graphs using a specialized declarative expression language called GReQL [24]. Rigi is based on graph transformations written in Tcl [91, 128].

Relational queries. Often, relational algebra engines abstract data out of entity-relation databases. Dali [56, 57] and ARMIN [58] use SQL queries to define grouping rules. Relational algebra defines a repeatable set of transformations such as abstraction or decomposition to create a particular architectural view. In PBS/SBS, Holt *et al.* propose the Grok relational expression calculator to reason about software facts [49]. Krikhaar presents a SAR approach based on an extension of relational algebra [30, 64].

Logic queries. Logic queries are powerful because of the underlying unification mechanism which allows us the writing of dense multi valued queries. Kramer and Prechelt [63], Wuyts [144], Gueneheuc [39] use Prolog queries to identify design patterns. Mens and Wuyts use Prolog as a meta programming language to extract intensional source-code views and relations in Intensive [87, 145]. Richner also chose a logic query based approach to reconstruct architectural views from static and dynamic facts [104].

Programs. Some approaches build analyses as plain object-oriented programs. For example, the analyses made in the Moose environment are performed as object-oriented programs that manipulate models representing the various inputs [20].

Lexical and structural queries. Some approaches are directly based on the lexical and structural information in the source code. Pinzger *et al.* state that some hot-spots clearly localize patterns in the source code and consider them as the starting point of SAR [100, 101]. To drive a pattern-supported architecture recovery, they introduce a pattern specification language and the Revealer tool. RM-Tool identifies architectural elements and relations using lexical queries [92, 93]. The Searchable Bookshelf is a typical example of supporting navigation via queries [113]. ArchVis [45] supports multiple inputs (files, programs, Acme information), works from static and dynamic information (program execution but also log files and network traffic), and provides different views to specific stakeholders (component, developer, manager views).

Investigation-based techniques. These techniques map high-level concepts with low-level ones. The considered high-level concepts cover a wide area from architectural descriptions and styles to design patterns and features. Explored approaches are:

Recognizers. ManSART [4, 43], ART [32], X-Ray [86] and ARM [40] are based on recognizers for architectural styles or patterns written in a query language. The tools then report the source code elements matching the recognized structures. More precisely, pattern definitions in ARM are progressively refined and finally transformed in SQL queries exploitable in Dali [56, 57].

Graph pattern matching. In ARM [40], pattern definitions can also be transformed into graph patterns to match a graph-based source code representation; this is similar to what is done in [110].

State engine. In DiscoTect state machines are defined to check architectural styles conformance [146]. A state engine tracks the system execution at run-time and outputs architectural events when the execution satisfies the state machine description.

Maps. SAR approaches based on the Reflexion Model [92, 93] use rules to map hypothesized high-level entities with source code entities. Since these Perl-like rules take plain source code as input, we could have classified the reflexion model in the *lexical and structural queries* group mentioned previously, but the real focus is on the mapping. SoFi [12] use naming conventions of files and folders to automatically group entities.

5.3. Quasi-automatic techniques

Purely automated SAR techniques do not exist. Reverse engineers must still steer the most automated approaches. Those often combine concept, dominance and cluster analysis techniques.

Concepts. Formal concept analysis is a branch of lattice theory used to identify design patterns [3], features [25, 38], or modules [111]. Tilley *et al.* [130] present a survey of works using formal concept analysis [5, 16, 108, 112, 115, 131, 136].

Clustering algorithms. Clustering algorithms identify groups of objects whose members are similar in some way. They have been used to produce software views of applications. To identify subsystems, Anquetil and Lethbridge

[1] Different of ArchView Pinzger's approach [99], though homonymous.

cluster files using naming conventions [2]. Some approaches automatically partition software products into cohesive clusters that are loosely interconnected [1, 79, 80, 90, 133, 140]. Clustering algorithms are also used to extract features from object interactions [109]. Koschke emphasizes the need to refine existing clustering techniques, first by combining them, and second by integrating the reverse engineer as a conformance supervisor of the reconstruction process [13, 62].

Dominance. In directed graph, a node D dominates a node N if all paths from a given root to N go through D. In software maintenance, dominance analysis identifies the related parts in an application [10, 14, 37]. In the context of software architecture extraction, adhering to Koschke's thesis, Trifu unifies cluster and dominance analysis techniques to recover architectural components in object-oriented legacy systems [133]. Similarly, Lundberg *et al.* outline a unified approach centered around dominance analysis [75]. On one hand, they demonstrate how dominance analysis identifies passive components. On the other hand, they state that dominance analysis is not sufficient to recover the complete architecture: this requires other techniques such as concept analysis to take component interactions into account.

6. SAR Outputs

SAR approaches result in different outputs, among which visual software views, architectures and conformance data, as well as analyses.

6.1. Visual software views

A lot of approaches offer architectural views or use visualizations as output. As we mentioned earlier, several tools such as Rigi [91, 128], Shrimp/Creole [121, 122], GraphViz [33] or CodeCrawler [70] are used to visualize graph representations of software views [31, 56, 62, 101, 106, 110]. Some authors propose open toolkits to build architectural extractors [74, 88, 127].

Classifying the outputs of the various visualization approaches is difficult and outside of the scope of this article, but we can still distinguish some groups: some visualization approaches present source code entities and group them as boxes using the visualization tools mentioned above [31, 56, 62, 101, 106, 110]. Some offer enhanced views that provide architectural information [77, 87, 99]. In this context some approaches improve their visualizations with 2D/3D [29, 69, 74, 81, 127]. Finally some approaches define dedicated tool support to represent architectural elements and layers; for example, the Software Architecture Browser is a graphical editor dedicated to navigation in layers [27]. Pacione proposed both the architecture-oriented visualization tool Vanessa, and a taxonomy in which he surveyed related tools [96].

Some SAR approaches focus on the behavior of software (Section 4). Hamou-Lhadj *et al.* surveyed trace visualization tools [42]. To offer multiple views of an application, it is interesting to combine static and dynamic analyses [45, 68, 104, 124]. For example, Shimba [124] combines static and dynamic information to produce high-level views of Java systems; it displays static information with Rigi [91, 128], and dynamic information as state diagrams. Both views are thus displayed separately, but the reverse engineers can constrain the abstraction of each view to the other one.

Table 3. SAR technique overview

Tools	Quasi-manual	Semi-automatic Abstr.	Semi-automatic Invest.	Quasi-auto.
Alborz [110]			gpm	auto
ArchView [99]		rel		
ArchVis [45]	cns	rel, prg		auto
ARES [26]				
ARM [40]	cns	gra, rel		
ARMIN [58]		gra		
ART [32]			rec	
Bauhaus [13, 25, 62]			rec, map	auto
Bunch [79, 90]				auto
Cacophony [28]				
Dali [56, 57]	cns	rel		
DiscoTect [146]			sta	
Focus [18, 84]	cns			
Gupro [24]		gra		
Intensive [87, 145]		log		
ManSART [4, 43]	cns		rec	
MAP [117]				
PBS/SBS [8, 31, 49, 113]		rel	map	
PuLSE/SAVE [61, 103]			map	
QADSAR [118, 119]				
Revealer [100, 101]		lex		
RMTool [92, 93]			map	
SARTool [30, 64]		rel		
SAVE [89, 94]			map	
Softwarenaut [77]	exp			
... with Hapax [67, 76, 77]		gra		
Symphony,Nimeta [106, 135]				
URCA [6]				auto
W4 [44]			map	
X-Ray [86]			rec	auto
— [7]				
— [51]				auto
— [75]				auto
— [97]	cns, exp			auto
— [132]		rel	map	

cns construction exp exploration gra graph queries
rel relational queries log logic queries prg programs
lex lexical queries rec recognizers gpm graph pattern matching
sta state engine map maps auto quasi-automatic

6.2. Architecture

Since SAR approaches focus on providing better understanding of the applications, they tend to present reconstructed architectural views to stakeholders. As the code evolves, some approaches focus on the co-evolution of the

reconstructed architecture: Intensive [87, 145] synchronizes the architecture with its implementation and highlights the differences due to evolution.

Iterative approaches based on the reflexion model [13, 61, 92, 105] make explicit the absences, convergences and divergences between the conceptual architecture and the architecture that results from mapping source code elements to architectural elements.

Architecture Description Languages (ADLs) have been proposed both to formally define architectures and to support architecture-centric development activities [82]. In the context of SAR, X-Ray [86] uses Darwin [78] to express reconstructed architectural views. Darwin was also extended by Eixelsberger *et al.* [26]. Acme [36] has ADL-like features and is used in DiscoTect [146]. Huang *et al.* specify architectures with the ABC ADL [51].

6.3. Conformance

Some approaches focus on determining the conformance of an application to a given architecture. We distinguish two kinds of architecture conformance: horizontal conformance between similar abstractions and vertical conformance between different abstraction levels.

Horizontal conformance is checked between two reconstructed views, or between a conceptual and a concrete architecture, or between a product line reference architecture and the architecture of a given product. For example, SAR approaches for product line migration identify commonalities and variabilities among products, like in MAP [117]. Sometimes, SAR compares a conceptual architecture with the reconstructed one [40, 132]. Sometimes, an architecture must conform to architectural rules or styles, as discussed in Nimeta [106], the SARTool tool [30, 64], Focus [18, 84] and DiscoTect [146].

Vertical conformance assesses whether the recontructed architecture conforms to the implementation. Both Reflexion Model-based [92, 93] and co-evolution-oriented [87, 145] approaches revolve around vertical conformance.

6.4. Analysis

Some approaches perform extra analysis on the extracted architecture to qualify it or to refine it further. Reverse engineers use modularity quality metrics either to iteratively assess current results and steer the process, or to get cues about reuse and possible system improvement [62, 110].

A few SAR approaches propose other analyses: ArchView [99] provides structural and evolutionary properties of a software application. Eixelsberger *et al.* in ARES [26], and Stoermer in QADSAR [118, 119] reconstruct software architectures to highlight properties like safety, concurrency, portability or other high-level statistics [51].

ARM [40], Revealer [100, 101], Alborz [110] highlight architectural patterns or orthogonal artifacts.

Table 4. SAR output overview

	vis	desc	horz	vert	ana
Alborz [110]	vis				ana
ArchView [99]	vis				
ArchVis [45]	vis	desc			
ARES [26]	vis	desc			ana
ARM [40]	vis				
ARMIN [58]	vis				ana
ART [32]	vis				
Bauhaus [13, 25, 62]	vis			vert	
Bunch [79, 90]	vis				
Cacophony [28]					
Dali [56, 57]	vis	desc			ana
DiscoTect [146]	vis	desc	horz	vert	
Focus [18, 84]	vis				
Gupro [24]	vis				
Intensive [87, 145]	vis				
ManSART [4, 43]	vis				
MAP [117]	vis				
PBS/SBS [8, 31, 49, 113]	vis				
PuLSE/SAVE [61, 103]	vis			vert	ana
QADSAR [118, 119]	vis				ana
Revealer [100, 101]	vis				
RMTool [92, 93]	vis			vert	
SARTool [30, 64]	vis		horz	vert	ana
SAVE [89, 94]	vis			vert	
Softwarenaut [77]	vis				
... with Hapax [67, 76, 77]	vis				
Symphony,Nimeta [106, 135]	vis		horz	vert	ana
URCA [6]	vis				
W4 [44]	vis			vert	ana
X-Ray [86]	vis	desc			
— [7]	vis		horz		
— [51]		desc	horz		ana
— [75]	vis				
— [97]	vis				
— [132]	vis			vert	

vis architecture visualization desc architecture description
horz horizontal conformance vert vertical conformance
ana analysis

7. Discussion and conclusion

Here are some general points that appeared to us at the light of this survey. A lot of approaches visualize software entities but few work from diverse information sources or even take advantage of having different kinds of information. Several times this paper stresses the need to provide stakeholders with a large variety of views at different levels of abstraction. SAR must be integrated in an environment that provides reverse engineers with views at different levels of abstraction and means to navigate horizontally and vertically. To fulfill this requirement, we state that a mechanism is required to express consistently viewpoints whatever the level of abstraction of the views they respectively describe. In this perspective, the metamodel-based SAR outlined by Favre is promising [28].

Lots of works focused on extracting design information

such as design patterns but stopped building on this knowledge up to the architectural level. Similarly few works bring together features and architectural information.

Because it is complex to extract architectural components from source code, those are often simply mapped to packages or files. Even if this practice is understandable, we think it limits and overloads the *component* term.

We see that few works really take into account architectural styles. That may be the result of having different communities working on architectural description languages and maintenance.

SAR is complex and time consuming. The iterative aspects of SAR imposed themselves as a key point to ensure a successful reconstruction. Now to reach a high-level of maturity in leading such an activity, we advocate that SAR has to support co-evolution and conformance checking mechanisms. Indeed both horizontal and vertical conformance help bringing all the recovered views face to face. That confrontation allows reverse engineers to refine views iteratively, to identify commonalities and variabilities among views (especially if they represent product line architectures), to lead impact analysis or still to update views when the system evolves.

Since successful systems are doomed to continually evolve and grow, SAR approaches should support co-evolution mechanisms to keep all recovered views synchronized with the source code. The logic-based approach of Intensive proved to be efficient in checking horizontal and vertical conformance and in allowing co-evolution [87, 145].

It is hard to classify research approaches in a complex field where the subject matter is as fuzzy as software architecture. Still this survey has provided an organization of the significant fundamental contributions made on software architecture reconstruction. To structure the paper, we followed the general process of SAR: what are the stakeholders' goals; how does the general reconstruction proceed; what are the available sources of information; based on this, which techniques can we apply, and finally what kind of knowledge does the process provide. We believe that software architecture reconstruction is still an important topic since it is crucial for the understanding of large industrial applications and their evolution.

References

[1] Anquetil and Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *WCRE*, 1999.

[2] Anquetil and Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11, 1999.

[3] Arévalo, Buchli, and Nierstrasz. Detecting implicit collaboration patterns. In *WCRE*, 2004.

[4] A.S.Yeh, Harris, and Chase. Manipulating recovered software architecture views. In *ICSE*, 1997.

[5] Bojic and Velasevic. Reverse Engineering of Use Case Realizations in UML. In *SAC*, 2000.

[6] Bojic and Velasevic. A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *CSMR*, 2000.

[7] Bowman and Holt. Software architecture recovery using conway's law. In *the Centre for Advanced Studies Conference, CASCON'98*, 1998.

[8] Bowman, Holt, and Brewster. Linux as a case study: its extracted software architecture. In *ICSE*, 1999.

[9] Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 1983.

[10] Burd and Munro. An initial approach towards measuring and characterizing software evolution. In *WCRE*, 1999.

[11] Buschmann, Meunier, Rohnert, Sommerlad, and Stad. *Pattern-Oriented Software Architecture — A System of Patterns*. Wiley, 1996.

[12] Carmichael, Tzerpos, and Holt. Design maintenance: Unexpected architectural interactions. In *ICSM*, 1995.

[13] Christl, Koschke, and Storey. Equipping the reflexion method with automated clustering. In *WCRE*, 2005.

[14] Cimitile and Visaggio. Software salvaging and the call dominance tree. *JSS*, 28, 1995.

[15] Conway. How do committees invent ? *Datamation*, 14(4), 1968.

[16] Dekel and Gil. Revealing class structure with concept lattices. In *WCRE*, 2003.

[17] Demeyer, Tichelaar, and Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, Univ. of Bern, 2001.

[18] Ding and Medvidovic. Focus: A light-weight, incremental approach to software architecture recovery and evolution. In *WICSA*, 2001.

[19] Ducasse and Demeyer, eds. *The FAMOOS Object-Oriented Reengineering Handbook*. Univ. of Bern, 1999.

[20] Ducasse, Gîrba, Lanza, and Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*. Franco Angeli, 2005.

[21] Ducasse, Lanza, and Bertuli. High-level polymetric views of condensed run-time information. In *CSMR*, 2004.

[22] Ducasse and Tichelaar. Dimensions of reengineering environment infrastructures. *Journal on Software Maintenance*, 15(5), 2003.

[23] Dueñas, Lopes de Oliveira, and de la Puente. Architecture recovery for software evolution. In *CSMR*, 1998.

[24] Ebert, Kullbach, Riediger, and Winter. GUPRO – generic understanding of programs, an overview. Fachberichte Informatik 7–2002, Universität Koblenz-Landau, 2002.

[25] Eisenbarth, Koschke, and Simon. Locating features in source code. *IEEE Computer*, 29(3), 2003.

[26] Eixelsberger, Ogris, Gall, and Bellay. Software architecture recovery of a program family. In *ICSE*, 1998.

[27] Erben and Löhr. Sab - the software architecture browser. In *VISSOFT*, 2005.

[28] Favre. CacOphoNy: Metamodel-driven software architecture reconstruction. In *WCRE*, 2004.

[29] Feijs and Jong. 3d visualization of software architectures. *Communications of the ACM*, 41(12), 1998.

[30] Feijs, Krikhaar, and van Ommering. A relational approach to support software architecture analysis. *Software – Practice and Experience*, 28(4), 1998.

[31] Finnigan, Holt, Kalas, Kerr, Kontogiannis, Mueller, Mylopoulos, Perelgut, Stanley, and Wong. The software bookshelf. *IBM Systems Journal*, 36(4), 1997.

[32] Fiutem, Antoniol, Tonella, and Merlo. Art: an architectural reverse engineering environment. *Journal of Software Maintenance: Research and Practice*, 11(5), 1999.

[33] Gansner and North. An open graph visualization system and its applications to software engineering. *Software Practice Experience*, 30(11), 2000.

[34] Garlan. Software architecture: a roadmap. In *ICSE – Future of SE Track*, 2000.

[35] Garlan, Allen, and Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), 1995.

[36] Garlan, Monroe, and Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, chapter 3. Cambridge University Press, 2000.

[37] Girard and Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *ICSM*, 1997.

[38] Greevy and Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, 2005.

[39] Guéhéneuc, Sahraoui, and Zaidi. Fingerprinting design patterns. In *WCRE*, 2004.

[40] Guo, Atlee, and Kazman. A software architecture reconstruction method. In *WICSA*, 1999.

[41] Hamou-Lhadj, Braun, Amyot, and Lethbridge. Recovering behavioral design models from execution traces. In *CSMR*, 2005.

[42] Hamou-Lhadj and Lethbridge. A survey of trace exploration tools and techniques. In *CASON*. IBM Press, 2004.

[43] Harris, Reubenstein, and Yeh. Reverse engineering to the architectural level. In *ICSE*. ACM, 1995.

[44] Hassan and Holt. Using development history sticky notes to understand software architecture. In *IWPC*, 2004.

[45] Hatch. *Software Architecture Visualisation*. PhD thesis, Research Institute in Software Engineering, Univ. of Durham, 2004.

[46] Heuzeroth, Holl, Högström, and Löwe. Automatic design pattern detection. In *IWPC*, 2003.

[47] Hofmeister, Nord, and Soni. *Applied Software Architecture*. Addison Wesley, 2000.

[48] Holt, Schürr, Sim, and Winter. Gxl: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2), 2006.

[49] Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE*, 1998.

[50] Holt. Sofware architecture as a shared mental model. In *ASERC Workshop on Software Architecture*, Univ. of Alberta, 2001.

[51] Huang, Mei, and Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering*, 13(2), 2006.

[52] IEEE. Ieee recommended practice for architectural description for software-intensive systems. Technical report, The Architecture Working Group of the Software Engineering Committee, 2000.

[53] Ivkovic and Godfrey. Enhancing domain-specific software architecture recovery. In *IWPC*, 2003.

[54] Jerding and Rugaber. Using visualization for architectural localization and extraction. In Baxter, Quilici, and Verhoef, eds., *WCRE*, 1997.

[55] Kazman and Carriere. View extraction and view fusion in architectural understanding. In *ICSR*, 1998.

[56] Kazman and Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 1999.

[57] Kazman, O'Brien, and Verhoef. Architecture reconstruction guidelines. CMU/SEI-2001-TR-026, Carnegie Mellon Univ., Software Engineering Institute, 2001.

[58] Kazman, O'Brien, and Verhoef. Architecture reconstruction guidelines, third edition. CMU/SEI-2002-TR-034, Carnegie Mellon Univ., Software Engineering Institute, 2003.

[59] Klein. *Sources of Power — How People Make Decisions*. Addison Wesley, 1999.

[60] Knodel, John, Ganesan, Pinzger, Usero, Arciniegas, and Riva. Asset recovery and their incorporation into product lines. In *WCRE*, 2005.

[61] Knodel, Muthig, Naab, and Lindvall. Static evaluation of software architectures. In *CSMR*, 2006.

[62] Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Univ. Stuttgart, 2000.

[63] Kramer and Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *WCRE*, 1996.

[64] Krikhaar. *Software Architecture Reconstruction*. PhD thesis, Univ. of Amsterdam, 1999.

[65] Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6), 1995.

[66] Kuhn, Ducasse, and Gîrba. Enriching reverse engineering with semantic clustering. In *WCRE*, 2005.

[67] Kuhn, Ducasse, and Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 2006.

[68] Lange and Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA*, New York NY, 1995.

[69] Langelier, Sahraoui, and Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE*, 2005.

[70] Lanza and Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE TSE*, 29(9), 2003.

[71] Lehman and Belady. *Program Evolution: Processes of Software Change*. London Academic Press, 1985.

[72] Lethbridge, Tichelaar, and Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, 2004.

[73] Li, Chu, Hu, Chen, and Yun. Architecture recovery and abstraction from the perspective of processes. In *WCRE*, 2005.

[74] Lowe and Panas. Rapid construction of software comprehension tools. In *Journal of Software Engineering and Knowledge Engineering*, 2005.

[75] Lundberg and Löwe. Architecture recovery by semi-automatic component identification. *Electronic Notes in Theoretical Computer Science*, 82(5), 2003.

[76] Lungu, Kuhn, Gîrba, and Lanza. Interactive exploration of semantic clusters. In *VISSOFT*, 2005.

[77] Lungu, Lanza, and Gîrba. Package patterns for visual architecture recovery. In *CSMR*, 2006.

[78] Magee, Dulay, Eisenbach, and Kramer. Specifying distributed software architectures. In *ESEC*, volume 989 of *LNCS*. Springer, 1995.

[79] Mancoridis and Mitchell. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *IWPC*, 1998.

[80] Maqbool and Babri. The weighted combined algorithm: A linkage algorithm for software clustering. In *CSMR*, 2004.

[81] Marcus, Feng, and Maletic. 3d representations for software visualization. In *SoftVis*. IEEE, 2003.

[82] Medvidovic and Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1), 2000.

[83] Medvidovic, Egyed, and Gruenbacher. Stemming architectural erosion by architectural discovery and recovery. In *International Workshop from Software Requirements to Architectures (STRAW)*, 2003.

[84] Medvidovic and Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2), 2006.

[85] Mendonca and Kramer. Requirements for an effective architecture recovery framework. In *ISAW-2)*, 1996.

[86] Mendonça and Kramer. An approach for recovering distributed system architectures. *Automated Software Engineering*, 8(3-4), 2001.

[87] Mens, Kellens, Pluquet, and Wuyts. Co-evolving code and design with intensional views – a case study. *Journal of Computer Languages, Systems and Structures*, 32(2), 2006.

[88] Meyer, Gîrba, and Lungu. Mondrian: An agile visualization framework. In *SoftVis*, 2006.

[89] Miodonski, Forster, Knodel, Lindvall, and Muthig. Evaluation of software architectures with eclipse. Technical report, Fraunhofer IESE, 2004.

[90] Mitchell and Mancoridis. On the automatic modularization of

software systems using the bunch tool. *IEEE TSE*, 32(3), 2006.
[91] Müller, Wong, and Tilley. Understanding software systems using reverse engineering technology. In Alagar and Missaoui, eds., *Object-Oriented Technology for Database and Software Systems*. World Scientific, 1995.
[92] Murphy, Notkin, and Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *FSE*, 1995.
[93] Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, Univ. of Washington, 1996.
[94] Naab. Evaluation of graphical elements and their adequacy for the visualization of software architectures. Master's thesis, Fraunhofer IESE, 2005.
[95] O'Brien, Stoermer, and Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical Report CMU/SEI-2002-TR-024, Carnegie Mellon Univ., 2002.
[96] Pacione. *A Novel Software Visualisation Model to Support Object-Oriented Program Comprehension*. PhD thesis, Univ. Strathclyde, 2005.
[97] Pashov and Riebisch. Using feature modelling for program comprehension and software architecture recovery. In *ECBS*, 2004.
[98] Perry and Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 1992.
[99] Pinzger. *ArchView – Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna Univ. of Technology, 2005.
[100] Pinzger, Fischer, Gall, and Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *WCRE*, 2002.
[101] Pinzger and Gall. Pattern-supported architecture recovery. In *IWPC*, 2002.
[102] Pinzger, Gall, and Fischer. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3), 2005.
[103] Pinzger, Gall, Girard, Knodel, Riva, Pasman, Broerse, and Wijnstra. Architecture recovery for product families. In *Int'l Workshop on Product Family Engineering*, LNCS 3014. Springer, 2004.
[104] Richner and Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM*, 1999.
[105] Richner and Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *ICSM*, 2002.
[106] Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Technical Univ. of Vienna, 2004.
[107] Riva and Rodriguez. Combining static and dynamic views for architecture reconstruction. In *CSMR*, 2002.
[108] Sahraoui, Melo, Lounis, and Dumont. Applying Concept Formation Methods to Object Identification in Procedural Code. In *ASE*, 1997.
[109] Salah and Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *ICSM*, 2004.
[110] Sartipi. *Software Architecture Recovery based on Pattern Matching*. PhD thesis, Univ. of Waterloo, Canada, 2003.
[111] Siff and Reps. Identifying Modules via Concept Analysis. In *ICSM*, 1997.
[112] Siff and Reps. Identifying modules via concept analysis. *IEEE TSE*, 25(6), 1999.
[113] Sim, Clarke, Holt, and Cox. Browsing and searching software architectures. In *ICSM*, 1999.
[114] Smolander, Hoikka, Isokallio, Kataikko, Mäkelä, and Kälviäinen. Required and optional viewpoints – what is included in software architecture? Technical report, Univ. Lappeenranta, 2001.
[115] Snelting and Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.
[116] Soni, Nord, and Hofmeister. Software architecture in industrial applications. In *ICSE*, 1995.
[117] Stoermer and O'Brien. Map - Mining architectures for product line evaluations. In *WICSA*, 2001.
[118] Stoermer, O'Brien, and Verhoef. Moving towards quality attribute driven software architecture reconstruction. In *WCRE*, 2003.
[119] Stoermer, Rowe, O'Brien, and Verhoef. Model-centric software architecture reconstruction. *Software — Practice and Experience*, 36(4), 2006.
[120] Storey, Fracchia, and Müller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Software Systems*, 44, 1999.
[121] Storey and Müller. Manipulating and Documenting Software Structures using SHriMP Views. In *ICSM*, 1995.
[122] Storey, Wong, and Müller. How do program understanding tools affect how programmers understand programs? In *WCRE*, 1997.
[123] Svetinovic and Godfrey. A lightweight architecture recovery process. In *WCRE*, 2001.
[124] Systä, Koskimies, and Müller. Shimba — an environment for reverse engineering Java software systems. *Software — Practice and Experience*, 1(1), 2001.
[125] Systä. On the relationships between static and dynamic models in reverse engineering java software. In *WCRE*, 1999.
[126] Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Univ. of Tampere, 2000.
[127] Telea, Maccari, and Riva. An open visualization toolkit for reverse architecting. In *IWPC*, 2002.
[128] Tilley. Domain-retargetable reverse engineering II: Personalised user interfaces. In *ICSM*, 1994.
[129] Tilley, Smith, and Paul. Towards a framework for program understanding. In *IWPC*, 1996.
[130] Tilley, Cole, Becker, and Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In Stumme, ed., *ICFCA*. Springer, 2003.
[131] Tonella. Concept Analysis for Module Restructuring. *IEEE TSE*, 27(4), 2001.
[132] Tran and Holt. Forward and reverse repair of software architecture. In *CASCON*, 1999.
[133] Trifu. *Using Cluster Analysis in the Architecture Recovery of Object-Oriented Systems*. PhD thesis, Univ. Karlsruhe, 2001.
[134] Tu and Godfrey. The build-time software architecture view. In *ICSM*, 2001.
[135] van Deursen, Hofmeister, Koschke, Moonen, and Riva. Symphony: View-driven software architecture reconstruction. In *WICSA*, 2004.
[136] van Deursen and Kuipers. Identifying objects using cluster and concept analysis. In *ICSE*, 1999.
[137] Vasconcelos and Werner. Software architecture recovery based on dynamic analysis. In *Brazilian Symposium on Software Engineering*, 2004.
[138] Walker, Murphy, Freeman-Benson, Wright, Swanson, and Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA*. ACM, 1998.
[139] Wendehals. Improving design pattern instance recognition by dynamic analysis. In *ICSE WODA*, 2003.
[140] Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *WCRE*, 1997.
[141] Wong. The rigi user's manual — version 5.4.4. Technical report, Univ. of Victoria, 1998.
[142] Wu, Sahraoui, and Valtchev. Program comprehension with dynamic recovery of code collaboration patterns and roles. In *CASCON*. IBM Press, 2004.
[143] Wu, Murray, Storey, and Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *WCRE*, 2004.
[144] Wuyts. Declarative reasoning about the structure object-oriented systems. In *TOOLS USA*, 1998.
[145] Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Univ. Brussel, 2001.
[146] Yan, Garlan, Schmerl, Aldrich, and Kazman. Discotect: A system for discovering architectures from running systems. In *ICSE*, 2004.