A Process-Oriented Software Architecture Reconstruction Taxonomy

Stéphane Ducasse, Damien Pollet, Loic Poyet

A Process-Oriented Software Architecture Reconstruction Taxonomy

Stéphane Ducasse Damien Pollet Loïc Poyet
LISTIC - Language and Software Evolution Group
Université de Savoie, France
Accepted to CSMR 2007

Abstract

To maintain and understand large applications, it is crucial to know their architecture. The first problem is that architectures are not explicitly represented in the code as classes and packages are. The second problem is that successful applications evolve over time so their architecture inevitably drifts. Reconstructing and checking whether the architecture is still valid is thus 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 first state of the art in software architecture reconstruction, with the desire to support the understanding of the field.

1. Introduction

Software architecture acts as a shared mental model of a system expressed at a high-level of abstraction [49]. By leaving details aside, this model plays a key role as a bridge between requirements and implementation [32]. It allows one to reason architecturally about a software application during the various steps of the software life cycle. According to Garlan [32], software architecture plays an important role in at least six aspects of software development: understanding, reuse, construction, evolution, analysis and management.

Software architecture is thus crucial for software development. The first problem is that architectures are not explicitly represented in the code as classes and packages are. The second problem is that successful software applications are doomed to continually evolve and grow [67]; and as a software application evolves and grows, so does its architecture. The conceptual architecture often becomes inaccurate with respect to the concrete architecture; this results in architectural erosion [75, 88], drift [88], mismatch [33], or chasm [96].

Software architecture reconstruction (SAR) is the reverse engineering process that aims at reconstructing viable architectural views of a software application. Krikhaar specified five SAR maturity levels [62]. While SAR is rarely an end in itself, it improves software development by providing high-level views of the investigated software application [43]. For example, these views help identify product line commonalities and variabilities [109] or check their conformance to the source code [82].

Several approaches and techniques have been proposed in the literature to support SAR. Mendonça *et al.* presented a first raw classification of SAR environments based on a few typical scenarios [78]. O'Brien *et al.* surveyed SAR practice needs and approaches [85]. Still, there is no comprehensive state of the art and it is often difficult to compare the approaches. This article presents a first state of the art in SAR, with the desire to help understand the field and to identify the current approaches, techniques and tools. The presented taxonomy 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 approach to take. The taxonomy takes into account the goals, the process, the inputs, the techniques and the outputs of SAR.

Section 2 first stresses some key vocabulary definitions and the challenges addressed in the field. Section 3 describes the criteria that we adopted in our taxonomy; sections 4 to 8 then cover each of these criteria, and we conclude.

2. SAR Challenges

Before going in more depth into the challenges of SAR, we feel the need to clarify the vocabulary.

2.1. Vocabulary

Software architecture. IEEE defines *software architecture* as *"the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and*

evolution"[52]; this is closely related to the definition of Shaw, Perry and Garlan [102, 35].

Architectural style. A software architecture often conforms to an *architectural style* that is a class of architectures, or a pattern of structural organization: *"a vocabulary of components and connector types, and a set of constraints on how they can be combined"* [102].

Architectural views and viewpoints. We can *view* a software architecture from several *viewpoints* since the different *system stakeholders* have different expectations or *concerns* about the system [64, 52]:

View: *"a representation of a whole system from the perspective of a related set of concerns."*

Viewpoint: *"a specification of the conventions for constructing and using a view. A pattern or a template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis."*

Conceptual architecture. It refers to the architecture that exists in human minds or in the software documentation [120, 96]. In the literature, this kind of architecture is also qualified as *idealized* [43], *intended* [132, 96], *as-designed* [57, 120] or *logical* [76].

Concrete architecture. It refers to the architecture that can be derived from source code [120, 96]. It is also known as the *as-implemented* [57, 96], *as-built* [120, 43], *realized* [132] or *physical* [76] architecture.

Software architecture reconstruction (SAR). SAR is the reverse engineering approach that aims at reconstructing viable architectural views of a software application. The literature uses several other terms to refer to SAR: *reverse architecting*, or architecture *extraction*, *mining*, *recovery* or *discovery*. The last two terms are more specific than the others [75]: *recovery* refers to a bottom-up process while *discovery* refers to a top-down process (see Section 5).

2.2. Challenges

SAR is a multidisciplinary activity which covers several research areas dealing with information processing: extraction, abstraction, modeling and presentation of the results.

On the one hand, human expertise is primordial to treat architectural concepts. Knowledge of business goals, requirements, product family reference architectures, or design constraints is useful to assist SAR. However, when we take human knowledge into consideration, several problems appear:

- Because of the experts turnover and the lack of complete, up-to-date documentation, the conceptual architecture in human minds is often obsolete, inaccurate, incomplete, or at an irrelevant abstraction level. SAR should take into account the quality of the information.
- When reconstructing an architecture, system stakeholders have various concerns such as performance, reliability, portability or reusability; SAR should support multiple architectural viewpoints.
- Reverse engineers are sometimes lost in the increasing software complexity. SAR needs to be interactive, iterative and parametrized.

On the other hand, source code is one of the few trustworthy reliable sources of information about the software application which contains its actual architecture. However, reconstructing the architecture from the source code raises several problems:

- The large amount of data held by the source code raises scalability issues.
- Since the considered systems are typically large, complex and long-living, SAR should handle development methods, languages and technologies that are often heterogeneous and sometimes interleaved.
- Architecture is not explicitly represented at the source code level. In addition, language concepts such as polymorphism, late-binding, delegation, or inheritance make it harder to analyze the code [130, 14]. How to identify the relevant information to reach an architectural level?
- The nature of software raises the questions of whether dynamic information should be extracted as the system is running, and then how do the behavioral aspects appear in the architecture.

To summarize this section, the major challenge of SAR is in abstracting, identifying and displaying higher-level views from lower-level and often heterogeneous information.

3. SAR Taxonomy Axes

Mendonça *et al.* [78] classified SAR environments and distinguished five families: filtering and clustering, compliance checking, analysers generators, program understanding and architecture recognition. O'Brien *et al.* surveyed SAR practice needs and approaches [85]. Gallagher *et al.* [30] proposed a framework to assess architectural visualization tools. Guéhéneuc *et al.* [38] proposed a comparative framework for design recovery tools. We propose a more elaborated classification based on the life-time of SAR presented in Figures 1 and 2): intended goals, followed processes, required inputs, used techniques and expected outputs.

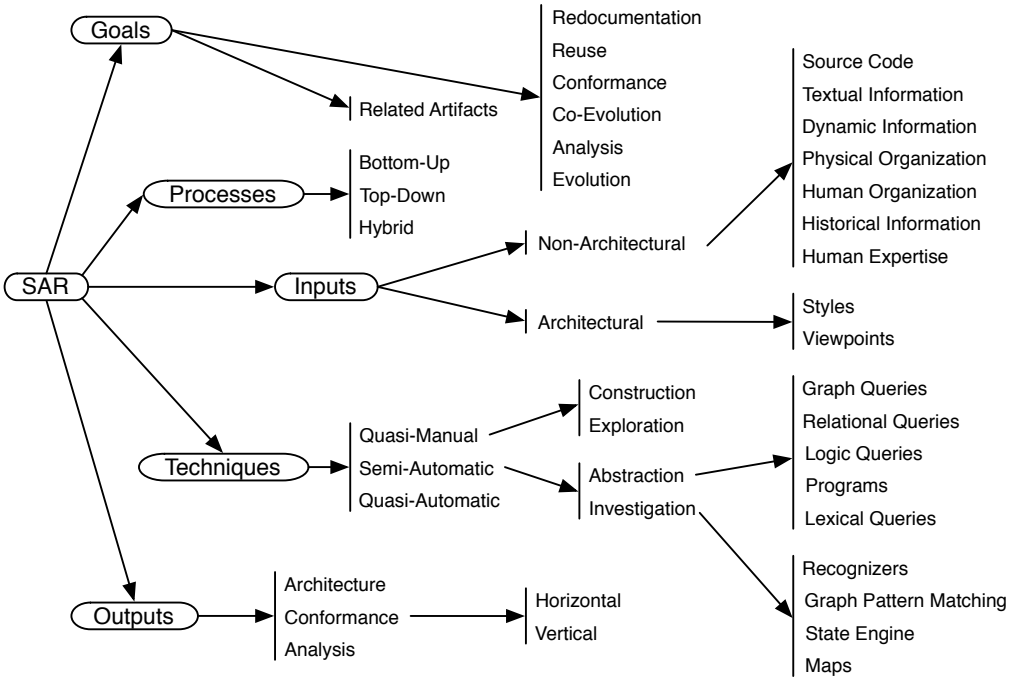

Figure 2. A process-oriented taxonomy

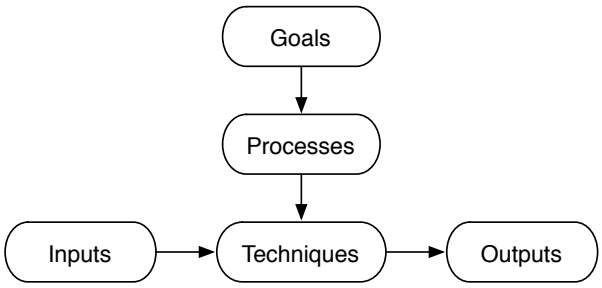

Figure 1. A process-oriented taxonomy flow

Goals. SAR is considered by the community as a proactive approach realized to answer stakeholder business goals. The reconstructed architecture is the basis for redocumentation, reuse investigation and product line migration, or implementation and architecture co-evolution. Some approaches do not extract the architecture itself but related and orthogonal artifacts that provide valuable extra information to engineers such as design patterns, roles or features.

Processes. We distinguish three kinds of SAR processes: bottom-up, top-down or hybrid.

Inputs. Most SAR approaches are based on source code information and human expertise. However, other kinds of information can be exploited: dynamic information or historical information. In addition, not all approaches support the specification and use of architectural styles and viewpoints which are the paramount of architecture.

Techniques. The research community has explored various techniques to reconstruct architecture that can be roughly classified according to their automation level.

Outputs. While all SAR approaches intend to provide architectural views, some of them however produce other valuable outputs like conformance data.

4. SAR Goals

We discuss now the goals of SAR as well as related artifacts.

4.1. Goals

Several authors categorized architecture roles in software development [32]; in particular, Kazman *et al.* have a pragmatic categorization of business goals [55]. In the context

of maintenance, a SAR process answers stakeholder business objectives. It must be considered as a proactive process realized for future forward engineering tasks. SAR approaches match various often interleaved intentions:

Redocumentation and understanding: The primary goal of SAR is to re-establish software abstractions. Recovered architectural views document software applications and help reverse engineers understand them. For instance, the software bookshelf introduced by Finningan *et al.* illustrates this goal [28]. Svetinovic *et al.* state that not only the recovered architecture is important, but also its rationale, *i.e.* why it is as it is [114]. They focus on the architecture rationale forces to recover the decisions made, their alternatives, and why each one was or was not chosen.

Reuse investigation and product line migration: Systematic reuse has not yet been achieved. Software product lines allow one to share commonalities among products while getting custom products. Architectural views are useful to identify commonalities and variabilities among products in a line [109, 93, 18, 23].

Conformance: To evolve a software application, it seems hazardous to use the conceptual architecture because it is often inaccurate with respect to the concrete one. In this case, SAR is a means to check conformance between the conceptual and the concrete architectures. Murphy *et al.* introduced the reflexion model and RM-Tool to bridge the gap between high-level architectural models and the system's source code [82]. Using SAR, reverse engineer can check conformance of the reconstructed architecture against rules or styles like in the SAR tool [62], Nimeta [96], DiscoTect [136], Focus [16, 76] and DAMRAM [75].

Co-evolution: Architecture and implementation are two levels of abstraction that evolve at different speeds. Ideally these abstractions should be synchronized to avoid architectural drift. Tran *et al.* propose a method to repair evolution anomalies between the conceptual and the concrete architectures, possibly altering either the conceptual architecture or the source code [120]. To dynamically maintain this synchronization, Wuyts promotes logic meta-programming [134], and Mens *et al.* exploit intensional source-code views and relations through Intensive [134, 80]; Favre promotes metaware [25]; Huang *et al.* propose a reflection mechanism too [51].

Analysis: In ArchView [89, 92], SAR and evolution analysis activities are interleaved. QADSAR is analysis-oriented too [110, 111]. An analysis framework steers a SAR framework so that it provides required architectural views to compute architectural quality analyses. This analysis assists stakeholders in their decision-making processes. Moreover, flexible SAR environments such as Dali [57, 84] or Gupro [19] support architectural analysis methods like SAAM [56] or ATAM [58] thanks to exportation facilities.

Evolution and maintenance: SAR is also a first step towards software evolution and maintenance. Focus is one of these approaches [16, 76]. Its strength is that the SAR scope is reduced to the system part which should evolve. Krikhaar *et al.* also introduced a two-phase approach for evolving architecture based on SAR and on change impact analyses [62, 63]. Huang *et al.* also consider SAR in an evolution and maintenance perspective [51].

4.2. Related and Orthogonal Artifacts

Some approaches do not extract the architecture in itself but architectural correlated or side-effect artifacts that crosscut or complement the architecture such as *design patterns*, *concerns*, *features*, *aspects*, or *roles* and *collaborations*. While such information is not directly related to the architecture (*i.e.* view points, architecture), it provides valuable extra information [4]. These approaches consider that higher level knowledge is necessary to extract valuable information at the architectural level and to improve the expressiveness of the reconstructed architectural views. Due to space limitation, this topic is only briefly surveyed in this paper.

It is well acknowledged that patterns play a key role in software engineering and this whatever their abstraction level [4, 8]. Some reverse engineering approaches consequently are based on design pattern identification activities [1, 3, 46, 128, 5, 39].

Concerns are the stakeholders' criterion for modularizing a software application into manageable and comprehensible parts [98, 12]. Features and aspects are more specific kinds of concerns. Features are considered in [131, 21, 87, 96, 36, 106] and aspect mining techniques in [10, 59, 83].

Source code artifacts interact together to fulfill software behaviors. Wu *et al.* highlight that source code is structured according a design in mind where software artifacts play conceptual roles inside collaborations [133]. The recovery of collaborations and roles was also explored in [95].

5. SAR Processes

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

5.1. Bottom-Up Processes

Bottom-up processes start with low-level knowledge to recover architecture. From source code models, they pro-

gressively raise the abstraction level until a high-level understanding of the application is reached [7, 112].

Also called architecture *recovery* processes, bottom-up processes are closely related to the well-known extract-abstract-present cycle described by Tilley *et al.* [119]. 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.

Examples. Several tools support a bottom-up process characterized by the extract-abstract-present metaphor: PBS [28], Rigi [81, 113], Gupro [19], Dali [57, 84].

As an example, Dali works as follows: (1) Heterogeneous low-level knowledge is extracted from the software implementation, fused and stored in a relational database. (2) Using Rigi, one visualizes and manually abstracts this information. (3) A reverse engineer can specify patterns using SQL queries and Perl expressions. The former selects a set of source model entities and the latter treats this set to abstract it. To summarize, Dali is a flexible workbench around a central model storage. Based on Dali, Guo *et al.* proposed ARM [40].

In Intensive, Mens *et al.* apply logic intension to group related source-code entities structurally in a view [134, 80]. Reverse engineers incrementally define views and relations by means of intensions specified as Smalltalk or Soul queries. Intensive classifies the views and displays consistencies and inconsistencies with the code and between architectural views.

Other bottom-up approaches for instance include ArchView [89, 92], Revealer [90, 91] and ARES [23, 22].

5.2. 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 [103, 112]. The term architecture *discovery* often describes such a process.

Examples. The Reflexion Model of Murphy *et al.* falls into this category [82]. First, the reverse engineer defines his high-level hypothesized views of the application. Second, he specifies how his view maps to the source model. Finally, RMTool identifies consistencies and inconsistencies. Like that, the reverse engineer iteratively interprets and computes successive reflexion models until satisfied.

Lungu *et al.* built both a method and a tool called Softwarenaut [71] to interactively explore hierarchical decompositions of software applications. Their method differs from other classical exploration tools: to construct an architectural view on the fly, they enhance the exploration process in guiding the reverse engineer towards the relevant hierarchical parts. They characterize packages based on their relation with the other ones and on their internal structure.

Categorizing such an approach shows the limit of a strict classification. The approach takes into account physical entities such as packages and does not check the conformance to predefined views as in the Reflexion Model. Still, we put it in this category since we considered that it flows from abstract to concrete entities: the exploration activity starts with the most abstract packages and iteratively open subpackages until to reach a relevant box and arrow view of the software application.

5.3. Hybrid Processes

Hybrid processes combine the previous two [112]. On the one hand, low-level knowledge is *abstracted* up using various techniques. On the other hand, high-level knowledge is *refined*. This kind of process is frequently used to stop architectural erosion by reconciling the conceptual and concrete architectures. Hybrid approaches often use hypothesis recognizers. Recognizer-based tools provide bottom-up reverse engineering strategies to support top-down exploration of architectural hypothesis. ManSART [43, 137], ART [29], X-ray [79], ARM [40] and DiscoTect [136] are examples of this approach. In ManSART, a top-down recognition engine maps a style-compliant conceptual view with a system overview which was defined using a visualization tool in a bottom-up fashion.

Examples. Sartipi implemented a pattern-based SAR approach in Alborz [100, 101]. The architecture reconstruction consists of two phases. During the first bottom-up phase, the source code is parsed, presented as a graph, then divided in cohesive graph regions using data mining techniques. This 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 previous graph regions 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 provided by Alborz.

Christl *et al.* present an evolution of the Reflexion Model [11]. 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 currently

unmapped concrete entities with concrete entities already mapped to hypothesized entities.

To assess the creation of product lines, Stoermer *et al.* introduce the MAP method [109]. 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 for instance include Focus [16, 76] and Nimeta [96].

6. SAR Inputs

SAR essentially works on source code representations. However, other kinds of information are sometimes considered such as dynamic information extracted from a system as it is running or historical data held by version control system repositories. In addition a few approaches take into account architectural elements such as styles or viewpoints as input to SAR. The current trend is to feed SAR with heterogeneous information of diverse abstraction levels.

6.1. Non-Architectural Inputs

Source Code Constructs. The source code is an omnipresent trustworthy source of information that most approaches consider. Some of them query directly the source code text like in RMTool [82]. However, most of them are not directly based on the source code but represent source code abstractions using different metamodels. These metamodels cope with the paradigm of the analyzed software. For instance, the language independent metamodel Famix is used for reverse engineering object-oriented applications [15]; its concepts include classes, methods, calls or accesses. Famix is used in ArchView [92, 89], Softwarenaut [71] and Nimeta [96]. Other metamodels such as the Dagstuhl Middle Metamodel [68] or GXL [50] have been proposed.

Symbolic Textual Information. Some approaches consider the symbolic information available in the comments [90, 91] or in the name of the methods [65].

Dynamic Information. Static information is often insufficient for SAR since it only provides a limited insight into the run-time nature of the analyzed software; dynamic information is more relevant to understand behavioral system properties. Some SAR approaches use dynamic information only [127, 136, 41] while others mix static and dynamic knowledge [54, 94, 97, 126, 69, 51, 89]. DiscoTect uses runtime events such as method calls, CPU utilization or network bandwidth consumption [136]. Huang *et al.* also considered this kind of information because it may inform reverse engineers on system security properties or system performance aspects.

Some works focus more on dynamic software information visualization [54, 116]; Hamou-Lhadj *et al.* present a deeper survey of this domain [42]. There are approaches based on dynamic information in areas adjacent to SAR: feature extraction [21, 99, 36], design pattern localization [128, 46], collaboration and role identification [95, 133]. Most of the time, dynamic information is generated from instrumented source code and use-cases.

Physical Organization. ManSART [43, 137] and Softwarenaut [71] take into account the structural organization of physical elements such as files, folders, or packages.

Human Organization. According to Conway [13]: *"Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations"*. Inspired by Conway's thesis, Bowman *et al.* use the developer organization to form an ownership architecture that helps reconstruct the software architecture [6].

Historical Information. Historical information is rarely used in SAR. Still ArchView is a recent approach that exploits source control system data and bug reports to analyze the evolution of recovered architectural views [89, 92]. To assist a reverse engineer to understand underlined dependency gaps in a reflexion model [82], Hassan *et al.* annotate entity dependencies with sticky notes. These sticky notes 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 validate results and to guide it. As Ivkovic *et al.* state [53], a SAR approach involves strategy and knowledge of the domain and the application itself. They propose to systematically update a knowledge base that would become an helpful collection of domain-specific architectural artifacts.

In current SAR approaches, it is frequent to specify a conceptual architecture [82, 44, 76]. To define this architecture, 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 6.2), or investigating orthogonal artifacts (Section 4.2).

6.2. Architectural Inputs

Architectural styles and viewpoints are the paramount of software architecture, therefore we analyzed whether SAR consider them.

Styles. Architectural styles are popular since like design patterns, they represent recurrent architectural situations [60]. They are valuable, expressive, and accepted abstractions for SAR and more generally for software understanding. Recognizing them is however a challenge because they basically span several architectural elements and can be implemented in various ways [91]. The question that turns up is whether SAR helps reverse engineers specify and extract architectural styles.

Examples. Ding *et al.* proposed to use architectural styles in Focus to infer a conceptual architecture that will be mapped to a concrete architecture extracted from the source code [16, 76].

Closely related to this work, Medvidovic *et al.* introduced an approach to stop architectural erosion. In a top-down process, requirements serve as high-level knowledge to discover the conceptual architecture [75]. 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. To select the most appropriate one, they characterize each architectural style according to several criteria. Their approach considers architectural styles as key design idioms since they capture a large number of design decisions, the rationale behind them, effective compositions of architectural elements, and system qualities that will likely result from the style's use.

Finally, DiscoTect considers architectural styles too [136]. It generates program traces by running the instrumented application. Then, it filters the traces and uses a state engine that incrementally recognizes interleaved execution patterns. In this way, DiscoTect reconstruct style-compliant architecture since by choosing a state machine, the reverse engineer defines and refines which hypothesized architectural style the tool should look for [114].

ManSART [43, 137] and MAP [109] are other style-based SAR approaches.

Viewpoints. As Holt states it: the architecture of a system acts as a mental model shared among stakeholders [49]. Since the stakeholders' interests in reverse engineering diverge, we must consider various viewpoints in SAR [52, 107]. Viewpoint catalogues were built to address this issue: the $4 + 1$ viewpoints of Kruchten [64]; the four viewpoints of Soni *et al.* [47, 108], the build-time viewpoint introduced by Tu *et al.* [122] or the implicit viewpoints inherent to the UML standard [123]. While most SAR approaches reconstruct architectural views only according to a single viewpoint or according to a few preselected ones, Smolander *et al.* highlight that viewpoints cannot be standardized but have to be selected or defined according to the environment and the situation [107]. O'Brien *et al.* notably present the View-Set Scenario pattern that helps determine which architecture views sufficiently describe the system and cover stakeholder needs [85].

Examples. The Symphony approach devised by van Deursen *et al.* aims at reconstructing software architecture using appropriate viewpoints [124]. Viewpoints are selected from a catalogue or defined if they don't exist. Moreover, they evolve throughout the process. Chosen viewpoints constrain SAR to provide architectural views compliant to stakeholders' expectations, ideally allowing an immediate use of these views. For example, Symphony authors highlight through four case studies some SAR motivations such as checking the conformance of family products to architectural rules. To do this they need to provide to reverse engineers architectural views according the viewpoints these reverse engineers typically use during design. Riva proposed a view-based SAR approach called Nimeta based on the Symphony one [96].

Favre outlines a generic SAR metamodel-driven approach called CacOphoNy [26]. Like Symphony, CacOphoNy recognizes the need to identify which viewpoints are relevant for stakeholder concerns and have to be considered in SAR. Contrary to Symphony, CacOphoNy states that metamodels are keys for representing viewpoints.

The QADSAR approach both reconstructs the architecture of a system and drives quality attribute analyses on it [110, 111]. To do this, QADSAR allows reverse engineers to formulate their interests in reconstructing the architecture by means of concrete quality attribute scenarios. This results in the definition of relevant architectural viewpoints.

ARES [23, 22] and SAR [62] also take viewpoints into account.

7. SAR Techniques

Techniques and the data they operate on are often correlated. For example, input information is represented respectively as facts [80] or graphs [19] to use logic or graph queries.

SAR approaches use different techniques that we classified according to their automation level: *quasi-manual,* the reverse engineer manually identifies architectural elements using a tool to assist him to understand 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 between the classifications are not clear-cut.

7.1. 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 SAR approaches; we considered two categories.

Construction-based Techniques. These techniques reconstruct the software architecture by manually abstracting low-level knowledge, thanks to interactive and expressive visualization tools — Rigi [81, 113], PBS [28], CodeCrawler [66].

Exploration-based Techniques. These techniques give reverse engineers an architectural view of the system by guiding them through the highest-level artifacts of the implementation, like in Softwarenaut [71]. The architectural view is then closely related to the developer's view. Instead of providing guidance, the SAB browser [24] allows reverse engineers to assign architectural layers to classes and to navigate the resulting architectural views.

Gallagher *et al.* [30] surveyed other architecture visualization tools: ArchView[1] [27], the Searchable Bookshelf [105], SoftArch [37], SoFi [9], LePUS [20] and ArchVis [45].

7.2. Semi-Automatic Techniques

Here the techniques automate repetitive aspects of SAR. The reverse engineer steers the iterative refinement or abstraction leading to the identification of architectural elements.

Abstraction-based Techniques. These techniques are based on technologies allowing reverse engineers to specify reusable abstraction rules and to execute them automatically. They aim to map low-level concepts with high-level concepts. Explored approaches are:

[1] Different of Pinzger's approach [89, 92], though homonymous.

Graph queries: Gupro queries graphs using a specialized declarative expression language called GReQL [19]. Rigi is based on graph transformations written in Tcl [81, 113].

Relational queries: Often, relational algebra engines abstract data of entity-relation databases. Dali uses SQL queries to define grouping rules [57, 84]. Relational algebra is used to define a repeatable set of transformations such as abstraction or decomposition for creating a particular architectural view. Holt *et al.* propose the Grok relational expression calculator to reason about software facts [48]. Krikhaar presents a SAR approach based on a Relational Algebra extension [62].

Logic queries: Mens and Wuyts uses Prolog as a meta programming language to extract intensional source-code views and relations in Intensive [134, 80]. Richner also chose a logic query based approach to reconstruct architectural views from static and dynamic facts [94].

Programs: Some approaches build analyses as programs. For example, the analyses made in the Moose environment are performed as object-oriented programs that manipulate models representing the various inputs [17].

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 hotspots clearly localize patterns in the source code and consider them as the starting point of SAR [90, 91]. To drive a pattern-supported architecture recovery, they introduce a pattern specification language and the Revealer tool.

Investigation-based Techniques. These techniques map high-level concepts with low-level concepts. The high-level concepts considered cover a wide area from architectural descriptions, styles, and patterns to design patterns, concerns, aspects, and features, that are orthogonal concepts to architecture and that we do not treat in this paper for space reasons (Section 4.2). Explored approaches are:

Recognizers. ManSART [43, 137], ART [29], X-ray [79] and ARM [40] are based on a set of architectural style or pattern recognizers written in a query language. More precisely, pattern definitions in ARM are progressively refined and finally transformed in SQL queries exploitable in Dali [57, 84].

Graph pattern matching. In ARM, pattern definitions can also be transformed into pattern graphs to match with a graph-based source code representation like in Alborz [100, 101].

State engine. In DiscoTect state machines are defined to check architectural styles conformance [136]. A state engine tracks at run-time the system execution and outputs architectural events when the execution satisfies

the state machine description.

Maps. SAR approaches based on the Reflexion Model [82] use rules to map hypothesized high-level entities with source code entities.

7.3. Quasi-Automatic Techniques

Pure automatic techniques failed in reconstructing software architectures, and even if current techniques tend towards an automatic process, reverse engineers must still steer them. Concept, dominance, and cluster analysis techniques are often combined.

The Bunch tool [73, 74] uses clustering algorithms to automatically partition software products into cohesive clusters that are loosely interconnected [129]. Clustering algorithms, based on hill climbing and genetic algorithms, are applied on module dependency graphs extracted from source code. The Bunch tool was extended to take into account human knowledge [74].

According to Xiao *et al.* [135], clustering techniques applied to dynamic analysis are as efficient as those applied to static analysis, and this research area is promising and unexplored.

The Bauhaus environment implements a wide number of clustering techniques [61, 21, 11]. 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.

Adhering to Koschke's thesis, Trifu unifies cluster and dominance analysis techniques for the recovery of architectural components in object-oriented legacy systems [121]. Similarly, Lundberg *et al.* outline a unified approach centered around dominance analysis [70]. 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: it requires other techniques such as concept analysis to take component interactions into account. Concept analysis techniques were explored by Siff *et al.* [104], van Deursen *et al.* [125], Arévalo [3, 2] or Eisenbarth *et al.* [21] and surveyed by Tilley *et al.* [118].

8. SAR Outputs

While most approaches focus on producing presentations of software architectures, some provide valuable additional information, like conformance data. It is not surprising since SAR outputs are clearly related with goals that lead to perform such an activity. In this section we highlight some key aspects of these outputs.

8.1. Architecture

Since SAR approaches are understanding-oriented, they tend to present reconstructed architectural views to stakeholders. As the code evolves some approaches focus on the co-evolution of the reconstructed architectures: Intensive [134, 80] synchronizes the architecture with its implementation; Focus [16, 76] or SAR [63] evolve the application.

Visualization. Rigi [81, 113] is widely used to visualize graph representations of software static views [28, 57, 18, 61, 91, 100, 96]. Rigi owes its success to its information manipulation features—since it was originally intended to reconstruct architectures—but also to its navigation capabilities and to its RSF exchange format. The SHriMP visualization technique enhances its navigation capabilities [113].

Several recent SAR tools [92, 80, 71] use CodeCrawler [66] and its underlying polymetric view technique. Riva [96] takes advantage of the strengths of different target visualization tools: SoftViz [117] and GraphViz [31] for graph browsing and manipulating, Hava [97] for static and dynamic information, and Rational Rose for UML diagrams. Focus [16, 76], Gupro [19], and the SWAGKit pipeline [28] respectively use Rational Rose, GraphViz also used in [74, 79], and LSEdit [115]. The SAB browser is a dedicated graphical editor to navigate layer [24]. Pacione proposed both a software-oriented visualization tool Vanessa, and a taxonomy in which he surveyed related tools [86].

As shown in Section 6, some SAR approaches focus on the behavior of software. Hamou-Lhadj *et al.* surveyed some of these tools dealing with visualization among others considerations [42].

Description. Architecture Description Languages (ADLs) have been proposed both to formally define architectures and to support architecture-centric development activities [77]. In the scope of this paper, Darwin [72] serves in X-ray [79] to define reconstructed architectural views. It was also extended by Eixelsberger *et al.* for their SAR approach [23, 22]. Acme [34] has ADL-like features and is used in DiscoTect [136]. Huang *et al.* specify architectures with the ABC ADL [51].

As said in Section 6.2, the notion of software architecture heavily depends on the stakeholders' interests. Since ADLs have difficulty in taking different viewpoints into account and focus on the module viewpoint, they are rarely used to express reconstructed architectural views. To drive SAR in CacOphoNy, Favre proposed to precisely define viewpoints using metamodels [26].

8.2. Conformance

We consider architecture conformance between similar abstraction levels (horizontal conformance) and between different abstraction levels (vertical conformance).

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 oriented towards a product line migration identify commonalities and variabilities among products, like in MAP [109]. Sometimes SAR requires to define a conceptual architecture and to compare it with the reconstructed concrete one [40, 120]. Sometimes, an architecture must conform to architectural rules or styles; this was discussed in Nimeta [96], the SAR tool [62], Focus [16, 76] and DAMRAM [75] and DiscoTect [136].

Vertical Conformance assesses whether the recontructed architecture conforms to the implementation. Both Reflexion Model-based [82] and co-evolution-oriented [80] approaches revolve around vertical conformance.

8.3. Analysis

Reverse engineers use modularity quality metrics either to iteratively assess current results and steer the process, or to get cues about reuse, system improvement Rigi [81, 113], Bauhaus [61, 21, 11] or Alborz [100, 101] provide such results.

A few SAR approaches are more analysis-oriented. Archview [89, 92] provides structural and evolutionary properties of a software application. Eixelsberger *et al.* in ARES [23, 22], and Stoermer in QADSAR [110, 111] reconstruct software architectures to highlight properties like safety, concurrency, portability or other high-level statistics [51].

Approaches taking architectural patterns or orthogonal artifacts into consideration highlight them. For instance, ARM [40], Revealer [90, 91] or Alborz [100, 101] highlight architectural patterns.

9. Conclusions

In this paper we surveyed research works in the field of software architecture reconstruction (SAR). 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. As usual it is hard to classify research works in a multidisciplinary domain, so in this paper we focused on the approaches most related to architecture reconstruction; as future work several related artifacts should be examined: design pattern identification, aspect mining. We also plan to identify lacks and future research axes in the field as well as providing an analysis of the pros and cons of the categorized approaches when it is possible.

Acknowledgments. We gratefully acknowledge the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872). We would like to thanks Tudor Girba and Orla Greevy for the early feedback on the paper.

References

[1] Antoniol, Fiutem, and Cristoforetti. Design pattern recovery in object-oriented software. In *IWPC*, pp. 153–160, 1998.

[2] Arévalo. *High Level Views in Object-Oriented Systems using Formal Concept Analysis*. PhD thesis, Univ. Berne, Berne, Jan. 2005.

[3] Arévalo, Buchli, and Nierstrasz. Detecting implicit collaboration patterns. In *WCRE*, pp. 122–131. IEEE CS, Nov. 2004.

[4] Beck and Johnson. Patterns generate architectures. In *ECOOP*, vol. 821 of *LNCS*, pp. 139–149, 1994.

[5] Beyer and Lewerentz. CrocoPat: A tool for efficient pattern recognition in large object-oriented programs. Tech. Report I-04/2003, Univ. Cottbus, 2003.

[6] Bowman and Holt. Software architecture recovery using conway's law. In *CASCON*, p. 6, 1998.

[7] Brooks. Towards a theory of the comprehension of computer programs. *Int'l Journal of Man-Machine Studies*, pp. 543–554, 1983.

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

[9] Carmichael, Tzerpos, and Holt. Design maintenance: Unexpected architectural interactions. vol. 00, p. 134. IEEE CS, 1995.

[10] Ceccato, Marin, Mens, Moonen, Tonella, and Tourwe. A qualitative comparison of three aspect mining techniques. vol. 00, pp. 13–22, 2005.

[11] Christl, Koschke, and Storey. Equipping the reflexion method with automated clustering. In *WCRE*, pp. 89–98.

[12] Coelho and Murphy. Presenting crosscutting structure with active models. In *AOSD*, pp. 158–168, 2006.

[13] Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[14] Demeyer, Ducasse, and Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In *WCRE*, 1999.

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

[16] Ding and Medvidovic. Focus: A light-weight, incremental

approach to software architecture recovery and evolution. In *WICSA*, pp. 191–, 2001.

[17] Ducasse, Gîrba, Lanza, and Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Sw. Maint. and Reeng.*, RCOST / Software Technology Series, pp. 55–71. 2005.

[18] Dueñas, de Oliveira, and de la Puente. Architecture recovery for software evolution. In *CSMR*, pp. 113–120, 1998.

[19] Ebert, Kullbach, Riediger, and Winter. GUPRO – generic understanding of programs, an overview. Tech. Report 7–2002, Univ. Koblenz-Landau, 2002.

[20] Eden. Visualization of object oriented architectures. In *ICSE*, May 2001.

[21] Eisenbarth, Koschke, and Simon. Locating Features in Source Code. *IEEE Computer*, 29(3):210–224, 2003.

[22] Eixelsberger and Gall. Describing software architectures by system structure and properties. In *COMPSAC*, pp. 106–111, 1998.

[23] Eixelsberger, Ogris, Gall, and Bellay. Software architecture recovery of a program family. In *ICSE*, pp. 508–511, 1998.

[24] Erben and Löhr. Sab - the software architecture browser. In *VISSOFT*. IEEE CS, Sept. 2005.

[25] Favre. Meta-model and model co-evolution within the 3d software space. In *ELISA*, 2003.

[26] Favre. CacOphoNy: Metamodel-driven software architecture reconstruction. In *WCRE*, pp. 204–213, 2004.

[27] Feijs and de Jong. 3d visualization of software architectures. vol. 41, pp. 72–78, 1998.

[28] Finnigan, Holt, Kalas, Kerr, Kontogiannis, Mueller, Mylopoulos, Perelgut, Stanley, and Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[29] Fiutem, Tonella, Antoniol, and Merlo. A cliché-based environment to support architectural reverse engineering. In *ICSM*. IEEE, 1996.

[30] Gallagher, Hatch, and Munro. A framework for software architecture visualisation assessment. In *VISSOFT*. IEEE CS, Sept. 2005.

[31] Gansner and North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

[32] Garlan. Software architecture: a roadmap. In *ICSE - Future of SE Track*, pp. 91–101, 2000.

[33] Garlan, Allen, and Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[34] Garlan, Monroe, and Wile. Acme: An architecture description interchange language. In *CASCON*, pp. 169–183, 1997.

[35] Garlan and Perry. Introduction to the special issue on software architecture. *IEEE TSE*, 21(4), 1995.

[36] Greevy and Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, pp. 314–323, 2005.

[37] Grundy and Hosking. High-level static and dynamic visualization of software architectures. vol. 00, p. 5. IEEE CS, 2000.

[38] Guéhéneuc, Mens, and Wuyts. A comparative framework for design recovery tools. In *CSMR*. IEEE CS, 2006.

[39] Guéhéneuc, Sahraoui, and Zaidi. Fingerprinting design patterns. In *WCRE*, pp. 172–181, 2004.

[40] Y. Guo, Atlee, and Kazman. A software architecture reconstruction method. In *WICSA*, pp. 15–34, 1999.

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

[42] Hamou-Lhadj and Lethbridge. A survey of trace exploration tools and techniques. In *CASCON*, pp. 42–55, 2004.

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

[44] Hassan and Holt. Using development history sticky notes to understand software architecture. *iwpc*, 00:183, 2004.

[45] Hatch. *Software Architecture Visualisation*. Ph.D. thesis, Univ. Durham, Mar. 2004.

[46] Heuzeroth, Holl, Hogstrom, and Lowe. Automatic design pattern detection. *iwpc*, 00:94, 2003.

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

[48] Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE*, pp. 210–219, 1998. ISBN: 0-8186-89-67-6.

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

[50] Holt, Schürr, Sim, and Winter. Gxl: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, 4 2006.

[51] Huang, Mei, and Yang. Runtime recovery and manipulation of software architecture of component-based systems. vol. 13, pp. 257–281, 2006.

[52] IEEE. Ieee recommended practice for architectural description for software-intensive systems. Tech. report, Arch. Work. Group of the Sw.Eng. Committee, 2000.

[53] Ivkovic and Godfrey. Enhancing domain-specific software architecture recovery. In *IWPC*, p. 266, 2003.

[54] Jerding and Rugaber. Using visualization for architectural localization and extraction. In *WCRE*, pp. 56–65, 1997.

[55] Kazman and Bass. Categorizing business goals for software architectures. CMU/SEI-2005-TR-021, CMU SEI, 2005.

[56] Kazman, Bass, Webb, and Abowd. Saam: A method for analyzing the properties of software architectures. In *ICSE*, pp. 81–90, 1994.

[57] Kazman and Carrière. Playing detective: Reconstructing software architecture from available evidence. *ASE*, 1999.

[58] Kazman, Klein, Barbacci, Longstaff, Lipson, and Carrière. The architecture tradeoff analysis method. In *ICECCS*, pp. 68–78, 1998.

[59] Kellens and Mens. A survey of aspect mining tools and techniques. Tech. Report INGI TR 2005-07, UCL, Belgium, 2005.

[60] Klein, Kazman, Bass, Carrière, Barbacci, and Lipson. Attribute-based architecture styles. In *WICSA*, pp. 225–244, 1999.

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

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

[63] Krikhaar, Postma, Sellink, Stroucken, and Verhoef. A two-phase process for software architecture improvement. In *ICSM*, p. 371, 1999.

[64] Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[65] Kuhn, Ducasse, and Gîrba. Enriching reverse engineering with semantic clustering. In *WCRE*, pp. 113–122, 2005.

[66] Lanza and Ducasse. Polymetric views—A lightweight visual approach to reverse engineering. vol. 29, pp. 782–795. IEEE CS, 2003.

[67] Lehman and Belady. *Program Evolution: Processes of Software Change*. 1985.

[68] Lethbridge, Tichelaar, and Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Elec. Notes in Theoretical Comp. Sci.*, vol. 94, pp. 7–18, 2004.

[69] Li, Chu, Hu, Chen, and Yun. Architecture recovery and abstraction from the perspective of processes. In *WCRE*, pp. 57–66, 2005.

[70] Lundberg and Löwe. Architecture recovery by semi-automatic component identification. *Electr. Notes Theor. Comput. Sci.*, 82(5), 2003.

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

[72] Magee, Dulay, Eisenbach, and Kramer. Specifying distributed software architectures. In *ESEC*, vol. 989 of *LNCS*, pp. 137–153. Springer-Verlag, Sept. 1995.

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

[74] Mancoridis, Mitchell, Chen, and Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *ICSM*, 1999.

[75] Medvidovic, Egyed, and Gruenbacher. Stemming architectural erosion by architectural discovery and recovery. In *STRAW*, 2003.

[76] Medvidovic and Jakobac. Using software evolution to focus architectural recovery. vol. 13, pp. 225–256, 2006.

[77] Medvidovic and Taylor. A classification and comparison framework for software architecture description languages. vol. 26, pp. 70–93, 2000.

[78] Mendonça and Kramer. Requirements for an effective architecture recovery framework. In *ISAW-2 and Viewpoints workshops*, pp. 101–105, 1996.

[79] Mendonça and Kramer. An approach for recovering distributed system architectures. vol. 8, pp. 311–354, 2001.

[80] 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):140–156, 2006.

[81] Müller, Wong, and Tilley. Understanding software systems using reverse engineering technology. In *Object-Oriented Technology for Database and Software Systems*, pp. 240–252. 1995.

[82] Murphy, Notkin, and Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT*, pp. 18–28, 1995.

[83] Nora, Said, and Fadila. A comparative classification of aspect mining approaches. *Journal of Computer Science 2*, 4:322–325, 2006.

[84] O'Brien and Stoermer. Architecture reconstruction case study. CMU/SEI-2003-TN-008, CMU SEI, 2003.

[85] O'Brien, Stoermer, and Verhoef. Software architecture reconstruction: Practice needs and current approaches. Cmu/sei-2002-tr-024, esc-tr-2002-024, CMU SEI, 2002.

[86] Pacione. *A Novel Software Visualisation Model to Support Object-Oriented Program Comprehension*. PhD thesis, Nov. 2005.

[87] Pashov and Riebisch. Using feature modeling for program comprehension and software architecture recovery. In *ECBS*, pp. 406–418, 2004.

[88] Perry and Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[89] Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Univ. Vienna, 2005.

[90] Pinzger, Fischer, Gall, and Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *WCRE*, pp. 170–178, 2002.

[91] Pinzger and Gall. Pattern-supported architecture recovery. In *IWPC*, pp. 53–61, 2002.

[92] Pinzger, Gall, Fischer, and Lanza. Visualizing multiple evolution metrics. In *SoftVis 2005*, pp. 67–75, 2005.

[93] Pinzger, Gall, Girard, Knodel, Riva, Pasman, Broerse, and Wijnstra. Architecture recovery for product families. In *PFE-5*, LNCS 3014, pp. 332–351, 2004.

[94] Richner and Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM*, pp. 13–22, 1999.

[95] Richner and Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *ICSM*, 2002.

[96] Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Univ. Vienna, 2004.

[97] Riva and Rodriguez. Combining static and dynamic views for architecture reconstruction. *CSMR*, 00, 2002.

[98] Robillard and Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE*, pp. 406–416, 2002.

[99] Salah and Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *ICSM*, 2004.

[100] Sartipi. *Software Architecture Recovery based on Pattern Matching*. PhD thesis, Univ. Waterloo, CA, 2003.

[101] Sartipi, Yee, and Safyallah. Alborz: An interactive toolkit to extract static and dynamic views of a software system. In *ICPC*, 2006. To appear.

[102] Shaw and Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[103] Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, 1980.

[104] Siff and Reps. Identifying Modules via Concept Analysis. In *ICSM*, pp. 170–179, 1997.

[105] Sim, Clarke, Holt, and Cox. Browsing and searching software architectures. vol. 00, p. 381, 1999.

[106] Smith and Munro. Identifying structural features of java

programs by analysing the interaction of classes at runtime. In *VISSOFT*. IEEE CS, Sept. 2005.

[107] Smolander, Hoikka, Isokallio, Kataikko, Mkel, and Klvinen. Required and optional viewpoints what is included in software architecture? Tech. report, Univ. Lappeenranta, 2001.

[108] Soni, Nord, and Hofmeister. Software architecture in industrial applications. In *ICSE*, pp. 196–207, 1995.

[109] Stoermer and O'Brien. Map - Mining architectures for product line evaluations. In *WICSA*, vol. 00, p. 35, 2001.

[110] Stoermer, O'Brien, and Verhoef. Moving towards quality attribute driven software architecture reconstruction. vol. 0, p. 46, 2003.

[111] Stoermer, Rowe, O'Brien, and Verhoef. Model-centric software architecture reconstruction. vol. 36, pp. 333–363, 2006.

[112] Storey, Fracchia, and Müller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Software Systems*, 44:171–185, 1999.

[113] Storey and Müller. Manipulating and Documenting Software Structures using SHriMP Views. In *ICSM*, pp. 275–284, 1995.

[114] Svetinovic and Godfrey. A lightweight architecture recovery process. In *WCRE*, Oct. 2001.

[115] Synytskyy, Holt, and Davis. Browsing software architectures with lsedit. In *IWPC*, pp. 176–178, 2005.

[116] Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Univ. Tampere, 2000.

[117] Telea, Maccari, and Riva. An open visualization toolkit for reverse architecting. *iwpc*, 00:3, 2002.

[118] Tilley, Cole, Becker, and Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In *ICFCA*, 2003.

[119] Tilley, Smith, and Paul. Towards a framework for program understanding. In *WPC*, p. 19, 1996.

[120] Tran and Holt. Forward and reverse repair of software architecture. In *CASCON*, 1999.

[121] Trifu. *Using Cluster Analysis in the Architecture Recovery of Object-Oriented Systems*. PhD thesis, Univ. Karlsruhe, 2001.

[122] Tu and Godfrey. The build-time software architecture view. In *ICSM*, pp. 398–407, 2001.

[123] Unified Modeling Language 1.5 spec. Tech. report, Mar. 2003.

[124] van Deursen, Hofmeister, Koschke, Moonen, and Riva. Symphony: View-driven software architecture reconstruction. In *WICSA*, pp. 122–134, 2004.

[125] van Deursen and Kuipers. Identifying Objects using Cluster and Concept Analysis. In *ICSE*, pp. 246–255, 1999.

[126] Vasconcelos and Werner. Software architecture recovery based on dynamic analysis. In *18th Brazilian Symp. on Softw. Eng.*, 2004.

[127] Walker, Murphy, Freeman-Benson, Wright, Swanson, and Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA*, pp. 271–283, 1998.

[128] Wendehals. Improving design pattern instance recognition by dynamic analysis. In *WODA*, 2003.

[129] Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *WCRE*, pp. 33–43, 1997.

[130] Wilde and Huitt. Maintenance Support for Object-Oriented Programs. *IEEE TSE*, SE-18(12):1038–1044, 1992.

[131] Wilde and Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[132] Woods, Carrière, and Kazman. The perils and joys of reconstructing architectures, 1999.

[133] Wu, Sahraoui, and Valtchev. Program comprehension with dynamic recovery of code collaboration patterns and roles. In *CASCON*, pp. 56–67, 2004.

[134] Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[135] Xiao and Tzerpos. Software clustering based on dynamic dependencies. *csmr*, 00:124–133, 2005.

[136] Yan, Garlan, Schmerl, Aldrich, and Kazman. Discotect: A system for discovering architectures from running systems. In *ICSE*, pp. 470–479, 2004.

[137] Yeh, Harris, and Chase. Manipulating recovered software architecture views. In *ICSE*, pp. 184–194, 1997.