Software Architecture Reconstruction: A Process-Oriented Taxonomy

Stéphane Ducasse, Damien Pollet

Software Architecture Reconstruction: a Process-Oriented Taxonomy

Accepted to TSE

Stéphane Ducasse Damien Pollet

***Abstract*—To maintain and understand large applications, it is important 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 software architecture reconstruction state of the art and it is often difficult to compare the approaches. This article presents a state of the art in software architecture reconstruction approaches.**

***Index Terms*—Software Architecture Reconstruction**

I. Introduction

SOFTWARE ARCHITECTURE acts as a shared mental model of a system expressed at a high-level of abstraction [66]. By leaving details aside, this model plays a key role as a bridge between requirements and implementation [48]. It allows you to reason architecturally about a software application during the various steps of the software life cycle. According to Garlan [48], 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 important for software development, but architectures do not have an explicit representation in most general purpose programming languages. Another problem is that successful software applications are doomed to continually evolve and grow [93]; 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 [105], drift [124], mismatch [49], or chasm [134].

Several approaches and techniques have been proposed in the literature to support software architecture *reconstruction* (SAR). Mendonça and Kramer [106] presented a first raw and simple classification of SAR environments based on a few typical scenarios (filtering and clustering, compliance checking, analysers generators, program understanding, architecture recognition). O'Brien et al. surveyed SAR practice needs and approaches [119]. Still, there is no comprehensive state of the art of SAR approaches and it is often difficult to compare the approaches.

This article presents a state of the art of software architecture reconstruction approaches. While it is a review on the research in SAR, we organized it from the perspective of a reverse engineer who wants to reconstruct the architecture of an existing application and would like to know which approaches to consider. We structure the field around the following axes: the goals, the process, the inputs, the techniques and the outputs of SAR approaches. In each axe, we classify both the most influential approaches and the original ones, with the goal to create a structured reference or map of the research field.

Stéphane Ducasse is with INRIA Lille Nord Europe, Damien Pollet is with the University of Lille 1 (France). {stephane.ducasse,damien.pollet}@inria.fr

Approach Selection

We extracted the information described in this taxonomy based only on published articles or on documents that are publicly available and trackable, such as PhDs and technical reports. We excluded industrial tools for accessibility reasons and we focused on the ideas presented.

We acknowledge that some of the information may be not totally correct since sometimes we had to interpret the description of tool or approach. To that regard, it should be noted that software architecture extraction approaches are often far from been really well-specified. In addition, as software architecture is a blurry concept by definition, it is hard to make clear distinctions. Therefore, the trade-off in this taxonomy is extent versus extreme precision with respect to the ideas and approaches. We organized the paper as a cartography rather than a comparison, because we believe that a taxonomy should structure the domain and provide a set of criteria, and that in the specific field of software architecture, the reader has to complement the information we put in perspective. Still we apply a rigorous selection process, as explained now.

In this paper, we select works in two steps. First, in addition to works that are extracting architectural information, we also consider approaches that do not specifically extract architecture but related artifacts such as design patterns, features, or roles, since they often crosscut or are low-level constituents of the architecture. We read 366 works (articles, PhD and reports) on architecture extraction and visualization and 76 on features, design patterns and aspects identification. Since there are often several articles around the same tool or approach, we selected the most significant ones, but it does not mean that the articles we do not cite are not interesting. In total we selected 181 articles, including some articles providing descriptions and definitions software architecture; the selection was driven by the excellence of the work, its originality, or its impact in the community —as perceived through the number of references in the literature.

We also 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 and architecture reconstruction.

In the second step, we support the comparison of the approaches with a table for each axis that structures this survey. In these tables, we only list works that are the most concerned about architectural extraction. For the sake of space, we consider two categories of works: first, important contributions, i.e., those which were pioneers or influenced subsequent works in the literature; and second, the original works taking a specific perspective or approach to the general problem —by original, we mean that the work did not gain a lot of following in the community, but is still interesting from the survey perspective. This second category is interesting because it broadens the SAR taxonomy.

A word about presentation. To avoid to have one single approach taking all the explanation space, we illustrate our classification with as different works as possible; therefore, the fact that we list a tool as an example does not necessary mean that it is the most cited or used.

We do not take into account works like ArchJava [1] that extend traditional languages to mix architectural and programming elements or other architectural description languages, since in such cases the architecture is not extracted from existing applications. We also exclude approaches proposing general methodology or guidelines that do not stress a specific point to support software architecture reconstruction [31, 82, 155].

Section II first stresses some key vocabulary definitions and the challenges of software architecture reconstruction. Section III describes the criteria that we adopted in our taxonomy. Sections IV to VIII then cover each of these criteria. Before concluding, Section IX surveys the extraction of artifacts related to software architecture such as design patterns and features.

II. SAR Challenges

Kruchten [87] presents a good overview of software architecture as a field and its history. Now before going into depth on the challenges of SAR, we feel the need to clarify the vocabulary.

A. 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"* [70]. This definition is closely related to Shaw and Garlan's [141].

Architectural style: A software architecture often conforms to an *architectural style*, that is a class of architectures or a pattern of structural organization. An architectural style is *"a vocabulary of components and connector types, and a set of constraints on how they can be combined"* [141].

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 [70, 88].

View. A view is *"a representation of a whole system from the perspective of a related set of concerns"* [70].

Viewpoint. A viewpoint is *"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"* [70].

Conceptual architecture: This term refers to the architecture that exists in human minds or in the software documentation [134, 162]. In the literature, conceptual architecture is also qualified as *idealized* [60], *intended* [134, 175], *as-designed* [74, 162] or *logical* [104].

Concrete architecture: This term refers to the architecture that is derived from source code [134, 162]. It is also known as the *as-implemented* [74, 134], *as-built* [60, 162], *realized* [175] or *physical* [104] architecture.

Software architecture reconstruction (SAR): Software architecture reconstruction is a 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 [105]: *recovery* refers to a bottom-up process while *discovery* refers to a top-down process (see Section V).

B. Challenges

One of the most obvious goals of SAR is to identify abstractions which represent architectural views or elements. In this context, two sources of information are considered: human expertise and program artifacts (e.g., source code and execution traces).

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 important to assist SAR. However, when we take human and business knowledge into consideration, several problems appear:

1) Because of the high rate of turnover among experts and the lack of complete up-to-date documentation, the conceptual architecture in human minds is often obsolete, inaccurate, incomplete, or at an inadequate abstraction level. SAR should take into account the quality of the information.
2) When reconstructing an architecture, system stakeholders have various concerns such as performance, reliability, portability or reusability; SAR should support multiple architectural viewpoints.
3) Reverse engineers sometimes get lost in the increasing complexity of software. SAR needs to be interactive, iterative and parameterizable [53].

On the other hand, source code and application execution are 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:

1) The approaches must scale to handle the large amount of data held by the source code.

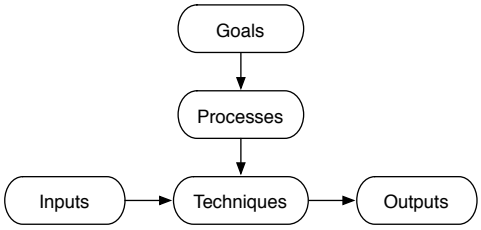

Fig. 1. The life-time flow of software architecture reconstruction, upon which we base this taxonomy

2) 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.
3) 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 [32, 170]. The extraction of a relevant architecture is then a difficult task.
4) The nature of software raises the questions of whether dynamic information should be extracted as the system is running, and then how do behavioral aspects appear in the architecture.

The major challenges of software architecture reconstruction are thus abstracting, identifying and presenting higher-level views from lower-level and often heterogeneous information.

III. Taxonomy Axes

Researchers already attempted to classify the field. Mendonça and Kramer [106] proposed a rough classification of SAR environments and distinguished five families based on the purpose of the approaches; they actually only define one criterium, with five values: filtering and clustering, compliance checking, analyzer generators, program understanding and architecture recognition. O'Brien et al. [119] presented some scenarios and approaches of SAR practice. Like us, they propose a pragmatic way to classify SAR approaches: they introduce recurring practice scenarios to characterize an approach: Viewset, enforced-architecture, quality-attribute-changes, common and variable artifacts, binary components, mixed languages. They then propose a technique axis to classify approaches along values of manual, manual with tool support, and query language. The two criteria roughly correspond to our *Goals* and *Techniques* axes. Gallagher et al. propose a framework to assess architectural visualization tools and compare a couple of tools [45]. Gueheneuc et al. present a comparative framework for design recovery tools and compared three approaches [56]. As a conclusion we can state that there is no deep and large survey of SAR.

We propose a deeper classification based on the life time of SAR approaches as depicted in Figures 1 and 2: intended goals, followed processes, required inputs, used techniques and expected outputs. Our taxonomy treats a larger number of approaches than the previous attempts. In particular, while focusing on SAR approaches, we analyze a broad range of works. We also put in context works related to program visualization, design patterns, and features extraction, since these works are related to the notion of architecture. We also mention some borderline works, but without comparing them in depth for space reasons.

Goals. SAR is considered by the community as a proactive approach to answer stakeholder business goals [31, 150]. 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.

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 exploit other architectural or non-architectural information sources such as dynamic information or historical information. In addition, not all approaches use architectural styles and viewpoints even though those are the paramount of 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.

IV. SAR Goals

To put in perspective the goals of SAR approaches, we briefly present the general goals of software architecture. According to Garlan, software architecture contributes to six main goals of software development [48]:

Understanding. Architectural views describe a software system at a level of abstraction high enough to understand its overall design, to reason about it and make decisions taking into account its design constraints, quality attributes, rationale, possible bottlenecks, etc.

Reuse. Architectural views strongly highlight candidates for reuse such as components, frameworks and patterns.

Construction. Architectural views are at a high-level of abstraction, allowing developers to focus their attention on the implementation of major components and relationships and iteratively to refine it.

Evolution. Architectural views make explicit the current constraints, and better expose how the software application is expected to evolve.

Analysis. Based on the high abstraction level of architectural views, new useful analyses can be performed, such as style conformance, dependence analysis, or quality attribute analysis.

Management. The clearer the view of the software system is, the more successful the development task will be.

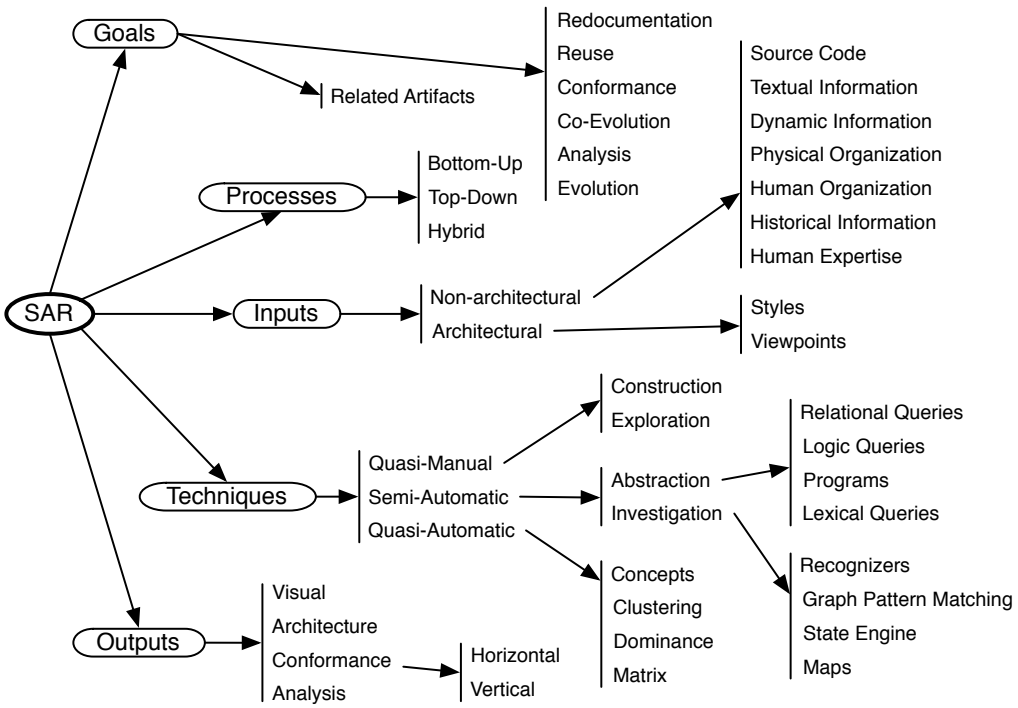

Fig. 2. A process-oriented taxonomy for SAR

A. Rearchitecting Goals

Several authors have categorized architecture roles in software development [48]; the roles involved in an architecture define the motivations for rearchitecting. In particular, Kazman and Bass have a pragmatic categorization of *business* goals [73] that motivate having an architecture in the first place. Similarly, in the context of maintenance, architecture reconstruction should answer the business objectives of stakeholders; it is a proactive process realized for future forward engineering tasks.

Knodel et al. identified ten distinct purposes or needs [83]; however, the purposes they present simultaneously are too narrow and do not cover all goals. This is why we do not use them for the present article. To classify SAR approaches in Table I, we grouped these purposes into six main goal categories refining the goals mentioned by Garlan [48].

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 [165]. For instance, the software bookshelf introduced by Finningan et al. illustrates this goal [12, 42, 67, 144]. Svetinotic and Godfrey state that not only the recovered architecture is important, but also its rationale, i.e., why it is as it is [155]. 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: Software product lines allow one to share commonalities among products while getting customized products. Architectural views are useful to identify commonalities and variabilities among products in a line [36, 129, 149]. SAR has also been used in the context of service-oriented architectures, to identify components from existing systems that can be converted into services [120].

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 RMTool to bridge the gap between high-level architectural models and the system's source code [114, 115]. Using SAR, reverse engineers can check conformance of the reconstructed architecture against rules or styles like in the SARTool [41, 86], Nimeta [134], Symphony [165], DiscoTect [180], Focus [24, 104] and DAMRAM [105].

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 and Holt propose a method to repair evolution anomalies between the conceptual and the concrete architectures, possibly altering either the conceptual architecture or the source code [162]. To dynamically maintain this synchronization, Wuyts uses logic meta-programming [179], and Mens et al. use intensional source-code views and relations through Intensive [108, 109, 179]; Favre [38] uses metaware (i.e., meta- and meta-meta-models); Huang et al. [69] use a reflection mechanism based on dynamic information.

Analysis: An analysis framework may steer a SAR framework so that it provides required architectural views to compute architectural quality analyses. Such analysis frameworks assist stakeholders in their decision-making processes. In ArchView [126], SAR and evolution analysis activities are interleaved. QADSAR is a tool that offers several analyses linked to threads, waiting points and performance properties [150, 151]. Moreover, flexible SAR environments such as Dali [74, 78],

	red	reus	cnf	coev	ana	evo
Alborz [139]	red					
ArchView [126]	red					
ArchVis [62]	red					
ARES [36]	red	reus				
ARM [57]	red					
ARMIN [79, 120]	red	reus			ana	
ART [43]	red					
Bauhaus [20, 35, 84]	red		cnf			
Bunch [100, 112]	red					evo
Cacophony [39]	red					
Dali [74, 78]	red				ana	
DiscoTect [180]	red		cnf			
Focus [24, 104]	red					evo
Gupro [33]	red					
Intensive [109, 179]	red		cnf	coev		
ManSART [60, 181]	red		cnf			
MAP [149]	red	reus				
PBS/SBS [12, 42, 67, 144]	red					
PuLSE/SAVE [83]	red	reus	cnf	coev	ana	
QADSAR [150, 151]	red				ana	
Revealer [127, 128]	red					
RMTool [114, 115]	red		cnf			
SARTool [41, 86]	red		cnf			evo
SAVE [111, 116]	red		cnf	coev		
Softwarenaut [97, 98]	red					
Symphony,Nimeta [134, 165]	red		cnf		ana	
URCA [10]	red					
W4 [61]	red		cnf			
X-Ray [107]	red					
— [11]	red		cnf			
— [69]	red			coev	ana	
— [96]	red					
— [123]	red					
— [162]	red		cnf	coev		

red re-documentation · reus reuse · cnf conformance
coev co-evolution · ana analysis · evo evolution

TABLE I
SAR GOAL OVERVIEW

	bottom-up	top-down	hybrid
Alborz [139]			hybrid
ArchView [126]	bottom-up		
ArchVis [62]	bottom-up		
ARES [36]	bottom-up		
ARM [57]			hybrid
ARMIN [79, 120]	bottom-up		
ART [43]			hybrid
Bauhaus [20, 35, 84]			hybrid
Bunch [100, 112]	bottom-up		
Cacophony [39]			hybrid
Dali [74, 78]	bottom-up		
DiscoTect [180]			hybrid
Focus [24, 104]			hybrid
Gupro [33]	bottom-up		
Intensive [109, 179]	bottom-up		
ManSART [60, 181]			hybrid
MAP [149]			hybrid
PBS/SBS [12, 42, 67, 144]			hybrid
PuLSE/SAVE [83]		top-down	
QADSAR [150, 151]			hybrid
Revealer [127, 128]	bottom-up		
RMTool [114, 115]		top-down	
SARTool [41, 86]	bottom-up		
SAVE [111, 116]		top-down	
Softwarenaut [97, 98]	bottom-up		
Symphony,Nimeta [134, 165]			hybrid
URCA [10]	bottom-up		
W4 [61]		top-down	
X-Ray [107]			hybrid
— [11]			hybrid
— [69]			hybrid
— [96]	bottom-up		
— [123]			hybrid
— [162]			hybrid

TABLE II
SAR PROCESS OVERVIEW

ARMIN [79, 120] or Gupro [33] support architectural analysis methods like SAAM [76] or ATAM [77] by exporting the extracted architectures to dedicated tools.

Evolution and maintenance: SAR is often a first step towards software evolution and maintenance. Here we use the term evolution to mean the study of the architecture as a tool to support application evolution and not the study the evolution itself. Understanding the inputs on which an approach is based is key to make this distinction: some approaches consider the history of a system to understand its evolution but not in the precise goal of directly supporting the system's evolution. Focus subscribes to that perspective; its strength is that the SAR scope is reduced to the system part which should evolve [24, 104]. Krikhaar et al. also introduced a two-phase approach to evolve architecture based on SAR and on change impact analyses [41, 86]. Huang et al. [69] also consider SAR in the perspective of evolution and maintenance.

B. Related and Orthogonal Artifacts

Some SAR approaches do not directly extract the architecture of an application but correlated artifacts that crosscut and complement the architecture. Such artifacts are *design patterns*, *features*, *aspects*, or *roles* and *collaborations*. While these artifacts are not the architecture itself (i.e., view points or architecture), they provide valuable information about it [8].

Patterns play a key role in software engineering at different levels of abstraction: architectural patterns, design patterns or idioms [8, 16]. Some reverse engineering approaches are thus based on design pattern identification [5, 6, 55, 63, 85, 178].

Features and aspects are also extracted from existing applications [35, 52, 123, 134, 171]. In the context of this paper we do not take aspect mining into account since a couple of surveys have already been published on the subject [18, 80, 118].

Roles and collaborations are important to object-oriented design: to achieve the program's task, objects collaborate with each other, each one playing a specific role [131]. However roles and collaborations are not explicit but buried into programs. Both Wu et al. [176] and Richner and Ducasse [133] support the extraction of roles and collaborations using dynamic information following the work of Lange and Nakamura [90].

V. SAR PROCESSES

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

A. Bottom-Up Processes

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

raise the abstraction level until a high-level understanding of the application is reached (see Figure 3) [14, 153].

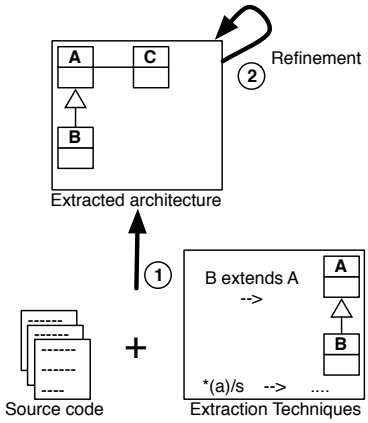

Fig. 3. A bottom-up process: from the source code, (1) views are extracted and (2) refined.

Also called architecture *recovery* processes, bottom-up processes are closely related to the well-known *extract-abstract-present* cycle described by Tilley et al. [159]. 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: The Dali tool by Kazman et al. [74, 78] supports a typical example of a bottom-up process: (1) Heterogeneous low-level knowledge is extracted from the software implementation, treated and stored in a relational database. (2) Using the Rigi visualization tool [113, 173], 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 [57] which focuses on design patterns conformance.

In Intensive, Mens et al. use logic intension to group related source-code entities in views that are robust to code changes [109, 179]. Reverse engineers incrementally define views and relations by means of 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 [92].

Lungu et al. built both a method and a tool called Softwarenaut [98] 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 the actions to get a better understanding of the software architecture.

Other bottom-up approaches include ArchView [126], Revealer [127, 128] and ARES [36], ARMIN [79, 120], Gupro [33]. We classify the works around PBS/SBS [12, 42, 67, 144] in this category, but since they consider conceptual architectures to steer the process, we could as well have classified them with the hybrid processes [12, 42, 67, 144].

B. 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 [17, 114, 153] (see Figure 4). The term architecture *discovery* often describes this process.

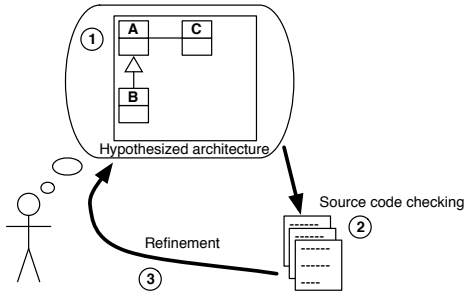

Fig. 4. A top-down process: (1) an hypothesized architecture is defined, (2) the architecture is checked against the source code, (3) the architecture is refined.

Examples: The Reflexion Model of Murphy et al. is a typical example of a top-down process [114, 115]. 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* (see Figure 5). 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 SAVE tool evaluates a given software architecture and its corresponding source code and points out the differences between these two artifacts in terms of convergences, divergences and absences [111]. The reflexion model offers a better support to express the conceptual architecture and the results of the process than the approach developed in SoFi [17]. The Reflexion Model influenced other works [20, 61, 83, 133, 162]. Not related to the Reflexion Model, Argo critics an architecture with high-level goals and at a high-level representation, however it is not clear how the architecture is effectively represented [136].

C. Hybrid Processes

Hybrid processes combine bottom-up with top-down processes [153, 165]. 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 (see Figure 6). Because hybrid processes reconcile the conceptual and concrete architectures, they are frequently used to stop architectural erosion [105, 124]. Hybrid approaches often use hypothesis recognizers that provide bottom-up reverse engineering strategies to support top-down exploration of architectural hypothesis [123].

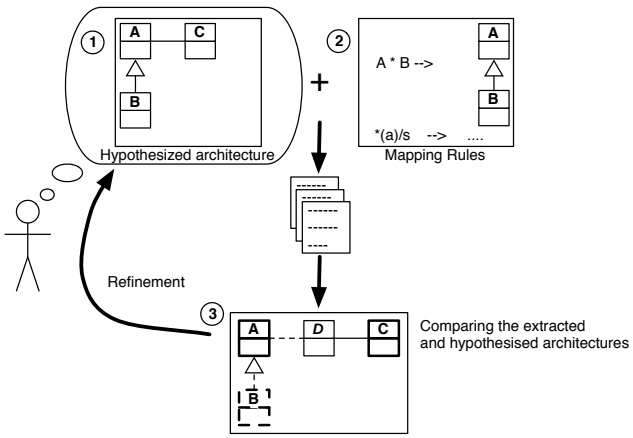

Fig. 5. The Reflexion Model, a top-down process: (1) an hypothesized architecture is defined, (2) rules map source entities to architectural elements, (3) RMTool compares the extracted and hypothesized architectures and the process iterates.

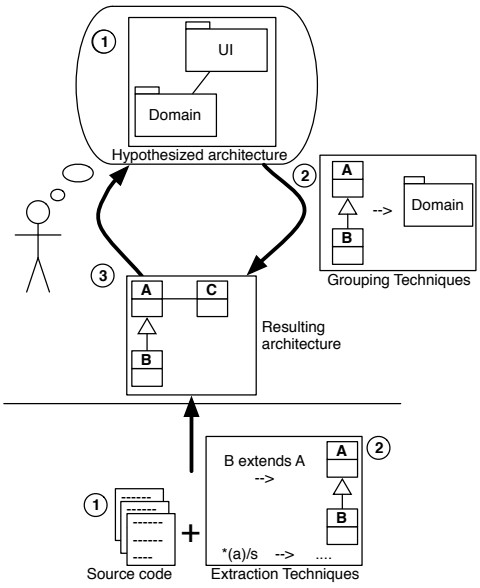

Fig. 6. An hybrid process.

Examples: Sartipi implements a pattern-based SAR approach in Alborz [139]. 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. [20] present an evolution of the Reflexion Model. 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 [149]. 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 [24, 104] and Nimeta [134], ManSART [60, 181], ART [43], X-Ray [107], ARM [57] and DiscoTect [180]. 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 [60, 181]. Pinzger et al. [129] present an approach to recover architecture for product families; they first determine the architectural views and concepts and then recover and assess the architecture using the Pulse-DSSA process [2].

As with any classification, the borders are fuzzy. For example, if the refinement step of a bottom-up approach is complex, we could categorize this approach as hybrid. We believe that this is not a real problem since the distinction still introduces important structure and flow to categorize the works. From Table II we can see that the three processes are represented in equal proportions.

VI. 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 diverse abstraction levels. We present first the non-architectural inputs, then the architectural inputs.

A. 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 [114, 115] or [127, 128]. However, most of them do not use 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 [23]; its concepts include classes, methods, calls or accesses. FAMIX is used in ArchView [126], Softwarenaut [98] and Nimeta [134]. Other metamodels such as the Dagstuhl Middle Metamodel [94] or GXL [65]

have been proposed with the same intent of abstracting the source code.

Symbolic Textual Information: Some approaches use the symbolic information available in the comments [127, 128] or in the method names [89, 102]. Anquetil and Lethbridge recover architecture from the source file names [4].

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 [90]. Some SAR approaches use dynamic information alone [180] while others mix static and dynamic knowledge [69, 95, 126, 132, 135, 166]. Walker et al. map dynamic information to architectural views [167]. Lots of approaches using dynamic information extract design views rather than architecture [58, 59, 82, 132, 156]. Huan 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 [69]. DiscoTect uses dynamic information too [180]. Li et al. uses run-time process information to derive architectural views [95]. Some works focus on dynamic software information visualization [27, 72, 156]; to get a more precise analysis of these, we refer the reader to the survey of Hamou-Lhadj and Lethbridge [59]. [10] use dynamic information extracted from use cases to identify packages and architectural views. Dynamic information is also used to identify features [35, 52, 137], design patterns [63, 168], or collaborations and roles [133, 176].

Physical Organization: The physical organization of applications in terms of files and folders often reveals architectural information. ManSART [60, 181] and Softwarenaut [98] 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 [91, 130, 177].

Human Organization: According to Conway's thesis: *"Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations"* [22]. It is then important to consider the influence of the human organization on the extracted architectures or views. Inspired by this, Bowman et al. use the developer organization to form an ownership architecture that helps stakeholders reconstruct the software architecture [11].

Historical Information: Historical information is rarely used in SAR. Wuyts [179] worked on the co-evolution between code and design. ArchView is a recent approach that exploits source control system data and bug reports to analyze the evolution of recovered architectural views [126]. Mens et al. analyze the evolution of extracted software views with Intensive [109, 179]. To assist a reverse engineer in understanding dependency gaps in a reflexion model, Hassan and Holt [61], Murphy et al. [114], Murphy [115] annotate entity dependencies with sticky notes. These sticky notes record dependency evolution and rationale with information extracted from version control systems. ArchEvo produces views of the evolution of modules that are extracted from source code entities [130].

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 results. To specify a conceptual architecture [61, 104, 114], 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 VI-B), or investigating orthogonal artifacts (Section IV-B). 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 and Godfrey [71] propose to systematically update a knowledge base that would become a helpful collection of domain-specific architectural artifacts.

B. Architectural Inputs

Architectural styles and viewpoints are the paramount of software architecture, we analyzed whether SAR approaches consider them as input to steer the extraction process. Some tools such as SAVE [111] take as input a mapping and architectural elements and apply the Reflexion Model (see Section V-B). Even if it is not exactly a SAR process, the Pulse approach produces a reference architecture by applying generic scenarios [2]. It works from domain models consisting of a decision model and generic work products, and was applied to statically evaluate architectures [82].

Styles: Architectural styles such as pipes and filters, layered system, data flow are popular because like design patterns, they represent recurrent architectural situations [16]. They are valuable, expressive, and accepted abstractions for SAR and more generally for software understanding. Examples of architectural styles are pipes and filters, blackboard, and layers.

Recognizing them is however a challenge because they span several architectural elements and can be implemented in various ways [127, 128]. 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 [24, 104].

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 [105]. 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 [180]. 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 [155].

ManSART [60, 181], ART [43] and MAP [149] are other SAR approaches taking architectural styles into account.

Viewpoints: The system architecture acts as a mental model shared among stakeholders [66]. Since the stakeholders' interests are diverse, viewpoints are important aspects that SAR may consider [70, 146]. Viewpoint catalogues were built to address this issue: the $4 + 1$ viewpoints of Kruchten [88]; the four viewpoints of Hofmeister et al. [64], Soni et al. [148], the build-time viewpoint introduced by Tu and Godfrey [164] or the implicit viewpoints inherent to the UML standard. Most SAR approaches reconstruct architectural views according only to a single 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 the situation [146]. O'Brien et al. present the View-Set Scenario pattern that helps one to determine which architectural views sufficiently describe the system and cover the stakeholders' needs [119].

Examples: The Symphony approach of van Deursen et al. aims at reconstructing software architecture using appropriate viewpoints [165]. Viewpoints are selected from a catalogue or defined if they don't exist, and they evolve throughout the process. They constrain SAR to provide architectural views that match the stakeholders' expectations, and ideally are immediately usable. The authors show how to define viewpoints step by step, and apply their approach on four case studies with different stakeholder goals. They provide architectural views to reverse engineers following the viewpoints these reverse engineers typically use during design phases. Riva proposed a view-based SAR approach called Nimeta based on Symphony [134]: Nimeta is a full SAR approach that uses the Symphony methodology to define viewpoints.

Favre outlines a generic SAR metamodel-driven approach called Cacophony [39]. 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: they specify the language that views have to conform to.

The QADSAR approach both reconstructs the architecture of a system and drives quality attribute analyses on it [150, 151]. To identify the relevant architectural viewpoints, reverse engineers formulate scenarios that highlight interesting quality attributes of the system. ARES [36] and SARTool [41, 86] also take viewpoints into account.

C. 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. advocate the fusion of multiple source of inputs to produce richer architectural views: for example, they produce interprocess communication and file access views [75]. Lange and Nakamura mix dynamic and static views to support design pattern extraction [90].

ArchVis [62] uses source code, dynamic information such as network log or messages sends and file structures.

	src	text	dyn	phys	org	hist	exp	style	viewp
Alborz [139]	src		dyn				exp		
ArchView [126]	src		dyn			hist	exp		
ArchVis [62]	src	text	dyn	phys				style	viewp
ARES [36]	src						exp		
ARM [57]	src						exp		
ARMIN [79, 120]	src			phys			exp		
ART [43]	src						exp	style	
Bauhaus [20, 35, 84]	src		dyn				exp		
Bunch [100, 112]	src						exp		
Cacophony [39]							exp		viewp
Dali [74, 78]	src						exp		
DiscoTect [180]	src		dyn				exp	style	
Focus [24, 104]	src						exp	style	
Gupro [33]	src						exp		
Intensive [109, 179]	src						exp		
ManSART [60, 181]	src			phys			exp	style	
MAP [149]	src						exp	style	
PBS/SBS [12, 42, 67, 144]	src			phys			exp		
PuLSE/SAVE [83]	src						exp		viewp
QADSAR [150, 151]	src						exp		viewp
Revealer [127, 128]	src	text					exp		
RMTool [114, 115]	src						exp		
SARTool [41, 86]	src						exp		viewp
SAVE [111, 116]	src						exp		
Softwarenaut [97, 98]	src	text		phys			exp		
Symphony,Nimeta [134, 165]			dyn				exp		viewp
URCA [10]	src		dyn				exp		
W4 [61]	src					hist	exp		
X-Ray [107]	src						exp		
— [11]	src				org	hist	exp		
— [69]	src		dyn					style	
— [96]	src						exp		
— [123]	src		dyn				exp	style	
— [162]	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

TABLE III
SAR INPUT OVERVIEW

Knodel et al. [82] discuss the combination of different information sources such as documents, source code and historical data. However it is not clear whether the approach was used in practice. Multiple inputs must be organized and Ivkovic and Godfrey propose a systematic way to organize application domain knowledge into a unified structure [71].

VII. SAR TECHNIQUES

There is a variety of formalisms used to represent, query and exchange the data representing applications [26, 50, 135]. A couple of exchange formats exist from simple textual tuples in RSF [173] or in TA [12, 42, 67, 144], to XML in GXL [33, 65, 134], or to CDIF in FAMIX [23]. The format may limit the merging or manipulation of the information it represents [26]. An important property of an exchange format is that it can be easily generated and used with simple tools [25].

SAR techniques are often correlated with the data they operate on: for example, Mens et al. express logic queries on facts [109, 179] while Ebert et al. perform queries on graphs [33]. 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 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 in the classification are not clear-cut and the categories are not mean to be exclusive. 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 IV synthesizes the classification of SAR techniques.

A. 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 [113, 173], CodeCrawler [92], Shrimp/Creole [152, 177], Verso [91], 3D [101] or GraphViz [47].

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 [98]. The architectural view is then closely related to the developer's view.

Instead of providing guidance, the SAB browser allows reverse engineers to assign architectural layers to classes and then to navigate the resulting architectural views [37].

ArchView[1] visualizes simple architectural elements and their relationships in 3D [40].

B. 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.

Abstraction-based Techniques: These techniques aim to map low-level concepts with high-level concepts. Reverse engineers specify reusable abstraction rules and execute them automatically. The following approaches were explored:

Relational queries. Often, relational algebra engines abstract data out of entity-relation databases. Dali uses SQL queries to define grouping rules [74, 78] [74]; so does ARMIN [79, 120]. Relational algebra defines a repeatable set of transformations such as abstraction or decomposition to create a particular architectural view. Gupro queries graphs using a specialized declarative expression language called GReQL [33]. Rigi is based on graph transformations written in Tcl [113, 173]. In PBS/SBS, Holt proposes the Grok relational expression calculator to reason about software facts [67]. It is based on Tarsky's relational algebra and as such is different from SQL-like queries. Krikhaar presents a SAR approach based on an extension of relational algebra [41, 86]. The ArchView abstraction algorithm combines relational algebra with metrics [126].

Logic queries. Logic queries are powerful because of the underlying unification mechanism which allows the writing of dense multi valued queries. Guéhéneuc et al. [55], Kramer and Prechelt [85], Wuyts [178] 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 [109, 179]. Richner and Ducasse also chose a logic query based approach to reconstruct architectural views from static and dynamic facts [132].

Programs. Some approaches build analyses as plain object-oriented programs. For example, the groupings made in the Moose environment are performed as object-oriented programs that manipulate models representing the various inputs [28].

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 [127, 128]. To drive a pattern-supported architecture recovery, they introduce a pattern specification language and the Revealer tool. RMTool identifies architectural elements and relations using lexical queries [114, 115]. The Searchable Bookshelf is a typical example of supporting navigation via queries [144]. Argo design critics uses Java predicates to automatically assess the current architecture of a system [136].

ArchVis supports multiple inputs (files, programs, Acme information), uses static and dynamic information (program execution but also log files and network traffic), and provides different views to specific stakeholders (component, developer, manager views) [62].

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 and styles to design patterns and features. Explored approaches are:

Recognizers. ManSART [60, 181], ART [43], X-Ray [107], ARM [57] and [44] 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 [74, 78]. The design patterns extraction approaches fit in this category (see Section IX).

Graph pattern matching. In ARM [57], pattern definitions can also be transformed into graphs pattern to match with a graph-based source code representation; this is similar to what Alborz [139] does.

State engine. In DiscoTect state machines are defined to check

[1]Different of ArchView Pinzger's approach [126], though homonymous.

architectural styles conformance [180]. 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 [114, 115] 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 intention here is really mapping. In SoFi, Carmichael et al. [17] use naming conventions of files and folders to automatically group entities.

C. Quasi-Automatic Techniques

Purely automated software architecture extraction techniques do not exist. Reverse engineers must still steer the most automated approaches. Approaches in this area often combine concept, dominance and cluster analysis techniques.

Concepts: Formal concept analysis is a branch of lattice theory used to identify design patterns [6], features [35, 52], or modules [143]. Tilley et al. [160] present a survey of work using formal concept analysis [10, 143, 161].

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 cluster files using naming conventions [4]. Some approaches automatically partition software products into cohesive clusters that are loosely interconnected [3, 100, 163, 169]. Clustering algorithms are also used to extract features from object interactions [137]. 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 [20, 84].

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 [21, 51]. 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 [163]. Similarly, Lundberg and Löwe outline a unified approach centered around dominance analysis [96]. On the 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.

Layers and Matrix: Often, applications are built with layers in mind: the lower layers should not communicate with the upper ones. An interesting approach for identifying cycles and layers in large applications is the Dependency Structure Matrix (DSM) [138, 154]. The Dependency Structure Matrix is adapted from the domain of process management [154] to analyze architectural dependencies in software [138]. Dependency structure matrixes show in a compact manner dependencies between source code entities such as classes and packages. Bril et al. use tree-cut based on a component connectivity metric to identify layers in the application dependency tree structure [13].

Tools	Quasi-manual	Semi-automatic Abstr.	Semi-automatic Invest.	Quasi-auto.
Alborz [139]			gpm	auto
ArchView [126]		rel		
ArchVis [62]	cns	rel, prg		auto
ARES [36]				
ARM [57]	cns	rel		
ARMIN [79, 120]		rel		
ART [43]			rec	
Bauhaus [20, 35, 84]			rec, map	auto
Bunch [100, 112]				auto
Cacophony [39]				
Dali [74, 78]	cns	rel		
DiscoTect [180]			sta	
Focus [24, 104]	cns			
Gupro [33]		rel		
Intensive [109, 179]		log		
ManSART [60, 181]	cns		rec	
MAP [149]	cns			
PBS/SBS [12, 42, 67, 144]		rel	map	
PuLSE/SAVE [83]			map	
QADSAR [150, 151]	cns	rel		
Revealer [127, 128]		lex		
RMTool [114, 115]			map	
SARTool [41, 86]		rel		
SAVE [111, 116]			map	
Softwarenaut [97, 98]	exp	rel		
Symphony,Nimeta [134, 165]				
URCA [10]				auto
W4 [61]			map	
X-Ray [107]			rec	auto
— [11]				
— [69]				auto
— [96]				auto
— [123]	cns, exp			auto
— [162]		rel	map	

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

TABLE IV
SAR TECHNIQUE OVERVIEW

VIII. SAR OUTPUTS

While most approaches focus on identifying and presenting software architectures, some provide valuable additional information, e.g., conformance of architecture and implementation. Indeed, goals and outputs are clearly related. In this section we highlight some points to further classify the approaches.

A. Visual Software Views

Several surveys of program visualization tools have been proposed. Gallagher et al. propose a framework to assess software visualizations around 7 key areas (static representation, dynamic representation, views, navigation, task support, implementation, visualization) and 31 features; however, they applied it to software extraction tools as well as UML case tools or notations

like LePUS [34]. Bassil and Keller propose a survey and analysis of software visualization [7], they also mention several other surveys. Here we do not reproduce such surveys but focus on the visualization as a possible output of the SAR process.

Supporting visualization tools: A lot of approaches offer (architectural) views or use visualizations as output [45] such as ArchVis [62]. As we mentioned earlier, several tools such as Rigi [113, 173], Shrimp/Creole [152, 177], GraphViz [47] or CodeCrawler [92] are used to visualize graph representations of software views [42, 74, 84, 127, 134, 139, 140]. Some authors propose open toolkits to build architectural extractors [122, 158] or scriptable visualizations [110].

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:

Architecture as boxes. Some visualization approaches present and group source code entities as boxes using the tools mentioned above [42, 74, 75, 84, 127, 134, 139]. For example, the Pulse approach [83] applied the SAVE tool to extract architectural views by grouping entities [111, 116].

Source entity visualization. Some tools focus on source code visualization or abstractions as opposed to true architectural entities. For example CodeCrawler [92], Distribution Map [29] and Package Blueprints [30] present condensed views of software source code entities. Similarly, Verso uses 3D to combine more information per entity [91].

Architectural Views. Some offer *enhanced views* that provide architectural information [98, 109, 126]. In this context some approaches improve their visualizations with 2D/3D [40, 101, 122, 158]. Erben and Lör 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 [37]. Grundi and Hosking propose the SoftArch tool which supports both static and dynamic visualisation of software architecture components at varying levels of abstraction. SoftArch copies, annotates, and animates static architectural views to provide developers with multiple, high-level execution architectural visualisations [53]. ArchVis [62] uses multiple sources and representations of architecture in to generate multiple views of software architecture. Pacione proposed both the architecture-oriented visualization tool Vanessa, and a taxonomy surveying related tools [121].

Orthogonally to this draft classification, and as shown in Section VI, some SAR approaches focus on the behavior of software and use execution traces. Hamou-Lhadj and Lethbridge [59] surveyed some of the tools supporting traces visualization. To offer multiple views of an application, it is interesting to combine static and dynamic analysis [27, 62, 90, 132, 157]. Program Explorer supports a navigation of design pattern elements using execution traces information [90]. For example, Shimba combines static and dynamic information to produce high-level views of Java systems; it displays static information with Rigi [113, 173], and dynamic information as state diagrams [157]. Both views are thus displayed separately, but the reverse engineers can constrain the abstraction of each view from the other one. Richner and Ducasse propose different views such as an invocation or instantiation relationships between high-level components [132].

B. Architecture

Since one an important goal of SAR approaches is to provide 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 [109, 179] synchronizes the architecture with its implementation and highlights the differences due to evolution.

Iterative approaches based on the reflexion model [20, 83, 114, 133] 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 [103]. In the context of SAR, X-Ray uses Darwin [99] to express reconstructed architectural views [107]. Darwin was also extended by Eixelsberger et al. [36] in ARES. Acme [50] has ADL-like features and is used in DiscoTect [180]. Huang et al. [69] specify architectures with the ABC ADL. They reconstruct architectural views and express them according to the ADL language in use to be coherent with an architecture-centric software development. In addition ADL features allow reverse engineers to give information in an ADL compliant format to improve the SAR process such as the layouts of architectural views that they have already produced.

C. Conformance

Some approaches focus on determining the conformance of an application to a given architecture [108]. 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 [149]. Sometimes, SAR requires to define a conceptual architecture and to compare it with the reconstructed one [57, 162]. Sometimes, an architecture must conform to architectural rules or styles; this was discussed in Nimeta [134], the SARTool tool [41, 86], Focus [24, 104] and DiscoTect [180]. Argo offers critics that comment the architecture or its potential problems [136]. Critics may also be low-level code critics such as abstract class wrong usage.

Vertical conformance assesses whether the reconstructed architecture conforms to the implementation. Both Reflexion Model-based [114, 115] and co-evolution-oriented [109, 179] approaches revolve around vertical conformance.

Alborz [139]	vis				ana
ArchView [126]	vis				
ArchVis [62]	vis	desc			
ARES [36]	vis	desc			ana
ARM [57]	vis				
ARMIN [79, 120]	vis				ana
ART [43]	vis				
Bauhaus [20, 35, 84]	vis			vert	
Bunch [100, 112]	vis				
Cacophony [39]					
Dali [74, 78]	vis	desc			ana
DiscoTect [180]	vis	desc	horz	vert	
Focus [24, 104]	vis				
Gupro [33]	vis				
Intensive [109, 179]	vis				
ManSART [60, 181]	vis				
MAP [149]	vis				
PBS/SBS [12, 42, 67, 144]	vis				
PuLSE/SAVE [83]	vis			vert	ana
QADSAR [150, 151]	vis				ana
Revealer [127, 128]	vis				
RMTool [114, 115]	vis			vert	
SARTool [41, 86]	vis		horz	vert	ana
SAVE [111, 116]	vis			vert	
Softwarenaut [97, 98]	vis				
Symphony,Nimeta [134, 165]	vis		horz	vert	ana
URCA [10]	vis				
W4 [61]	vis			vert	ana
X-Ray [107]	vis	desc			
— [11]	vis		horz		
— [69]		desc	horz		ana
— [96]	vis				
— [123]	vis				
— [162]	vis			vert	

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

TABLE V
SAR OUTPUT OVERVIEW

D. 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 [84, 139].

A few SAR approaches propose other analyses: ArchView [126] provides structural and evolutionary views of a software application. Eixelsberger et al. [36] in ARES, and Stoermer in QADSAR [150, 151] reconstruct software architectures to highlight properties like safety, concurrency, portability or other high-level statistics [69].

Finally, some approaches highlight architectural patterns or orthogonal artifacts: ARM [57], Revealer [127, 128] or Alborz [139].

IX. ORTHOGONAL OR RELATED ABSTRACTIONS

A large body of work focuses on extracting design or on reverse engineering applications. It is difficult to clearly separate these approaches from SAR since architecture has many forms and design information is important to characterize architecture. These approaches focus on identifying artifacts that either support the architecture, such as design patterns [8], or crosscut the architecture, such as features and roles. These related artifacts convey important information about the architecture; this is why we included them in this survey in a section of their own.

A. Design Patterns

Design patterns are important abstractions in programming and designing applications because they create a common vocabulary [46]. A design pattern highlights a recurring problem that arises in a specific design context, and discusses the possible solutions.

Beck and Johnson derive an architecture from a set of patterns [8]. Deducing an architecture from patterns records the design decisions that were made, and hints at their underlying motivations. Buschman et al. mention that patterns are useful mental building-blocks which compose and document the architecture [16]. Patterns span several levels of abstraction from architecture through design to language, and they are interwoven with each other. Architectural patterns or styles express high level fundamental organizations of systems; design patterns describe medium level structures of communicating components; language patterns or idioms present low level aspects of programming languages. For all these reasons researchers have been drawing considerable attention onto design pattern identification [90]. Gueheneuc, Mens and Wuyts propose a framework to compare design recovery tools [56].

Shull et al. propose a method to manually identify workable domain-specific design patterns and create customized catalogs of the identified patterns [142]. Brown automatically identifies design patterns using the reflective capabilities of Smalltalk [15]. Keller et al. promote pattern analysis as well as human expertise to extract design pattern [81]. Bergenti and Poggi provide critiques about the design patterns identified in UML documents [9]. Philippow et al. promote a design pattern-based approach to reconstruct the reference architecture of a product line [125].

Several approaches use Prolog to represent and query source code [85, 178]. Design patterns are then represented as logic queries. Lange et al. represents both static and dynamic information as logic facts to generate interactive design views and help understanding frameworks [90]. To extract design patterns that are based on specific interactions among the pattern participants, like Chain of Responsibility, researchers investigated dynamic analysis [63, 168]. One of the main problems in pattern identification is the search space. To reduce it, Wendehals [168] combines static and dynamic analyzes, the first one reducing the search space of the second one: the static analysis searches for sets of candidates that respect the static structure of the design pattern, while the dynamic analysis monitors candidates and checks whether the observed interactions satisfy the behavioral rules of the design patterns [63]. Gueheneuc et al. used explanation-based constraint programming to report problems when failing to identify design patterns [54]. Antoniol et al. propose a multi stage reduction strategy: software metrics and structural properties computed on design patterns become constraints that design pattern candidates must satisfy [5].

Guéhéneuc et al. [55] reduces the search space using metrics to define design pattern fingerprints of the design pattern participants. A design pattern has several design variants and can be implemented in different ways; Niere et al. [117] overcomes both problems with fuzzy logic, and Wendehals [168] rates instance candidates with fuzzy values to support inexact mismatch.

B. Features and Aspects

A key to software understanding is to locate source code entities according to the concerns they address. The decomposition of product families is often driven by the product features [145], and aspects represent crosscutting abstractions. While they are not directly related to software architecture, features often relate to functional requirements, while aspects often relate to non-functional requirements or to activity crosscutting a system; as such, both features and aspects provide interesting alternate views on the architecture of an application. However, these concerns are not explicitly linked to source code entities; in fact, they often crosscut the system's physical decomposition, scattered and tangled throughout its artifacts. Recovering crosscutting concerns is thus an active research area, but nowadays, researchers essentially focus their attention on mining concerns and rarely link their works with SAR, even though aggregating source code entities around the concerns they address could be a useful means of abstraction for SAR.

Features: According to Eisenbarth et al. [35] a feature is *"an observable behavior of a system that can be triggered by a user"* and a computational unit is *"an executable part of a system"*. A feature in the minds of reverse engineers is implemented through several computational units in the source code. To understand how a set of features is implemented, one must identify the computational units that contribute to these features and optionally the way they interact together. Features are high level knowledge while computational units are low level knowledge. More generally, features acts as a bridge between the requirements and the architecture of the system [123]. Therefore, a feature view improves software understanding by mapping functional requirements in the minds of reverse engineers with architectural elements and indirectly with source code entities. A feature view could help SAR by hiding implementation details around features.

The Software Reconnaissance method is a promising approach in the feature location field [171]. To identify the computational units related to a given feature, this method compares computational units invoked by different scenarios which trigger or not this feature. In a similar way, Wong et al. proposed an approach that analyses execution slices of different scenarios [174]. Chen et al. outlined a human-guided approach [19, 172]. Assisted by a tool, a reverse engineer explores a statically derived dependency graph and iteratively decides whether each considered computational unit is relevant to the feature or not.

Eisenbarth et al. combine static and dynamic analyses to derive the map linking features with computational units [35]. Using concept analysis, they obtain a map of relationships between features and computational units; this map is subject to human interpretation. Finally, they refine the map by deriving more relevant computational unit sets using static analysis such as dominance analysis. Salah and Mancoridis derive a feature map from object interactions; their method progressively raises the abstraction level from object interactions to feature interactions [137]. Greevy and Ducasse characterize features and computational units according to two complementary perspectives: A feature perspective and a computational unit perspective [52]. The approach allows for instance a reverse engineer to know how some computational units participate at the realization of a given feature.

Pashov and Riebisch promote the use of feature modeling to improve SAR [123]. Their feature-oriented approach iteratively reconstructs the architecture by establishing and verifying functional and architectural hypotheses. These hypotheses link features, architectural elements and source code entities in cross-referencing tables which are verified iteratively. Sochos et al. propose a method to offer a stronger mapping between features and the architecture based on a series of transformations on the initial product line feature. Architectural components are derived during the transformations and encapsulate the business logic of each transformed feature [147].

Aspects: Aspect mining receives a lot of attention currently. As said above, concerns often crosscut the implementation; aspect mining is the reverse engineering process which aims to find and isolate these crosscutting concerns. It is mainly explored to better understand a piece of software or to refactor it in an aspect-oriented one. Since there are already several surveys of the subject [18, 80, 118], we do not cover it here. It is however worth to mention that there are no approaches linking mining aspects with architecture extraction.

C. Collaborations and Roles

To understand an object-oriented application, one must understand the collaborations and the roles that objects play [131, 176]. Collaborations are goal-oriented interactions between participants, while roles describe the participants' responsibilities in a collaboration. Richner and Ducasse [133] focused on recovering collaborations and roles of objects and indirectly of classes. Hondt [68] proposes collaboration contracts as a basis to control the evolution of collaborations.

Some approaches deduce class collaborations by visualizing object interactions [59]. Richner et al. propose an approach to recover collaborations and roles that does however not rely on visualization techniques; they work from both dynamic information and human expertise [133]. Pattern matching tools extract collaboration patterns from execution traces that record method invocation information. To only focus on relevant class collaborations and roles, reverse engineers then steer the process through querying and visualization facilities. Wu et al. applied a closely related approach to procedural legacy systems [176].

X. Discussion

Here are some general points that appeared to us out of this survey. A lot of approaches visualize software entities but few work from diverse information or even take advantage of having different kinds of information. Several times this paper

stresses the need to provide a large variety of views at different levels of abstraction. We advocate that viewpoints should be defined consistently. SAR must integrate 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 [39] is promising.

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 term *component*.

We see that few works really take into account architectural styles. This 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 mechanisms. Indeed both horizontal and vertical conformance help the reverse engineer to bring all the recovered views face to face. This confrontation allows reverse engineers to refine views iteratively, to identify commonalities and variabilities among views (especially if they represent product lines architecture), 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 [109, 179].

XI. Conclusion

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 within 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 is still an important topic since it is a key abstraction for the understanding of large industrial applications and their evolutions.

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 thank Tudor Girba and Orla Greevy for the early feedback on the paper, and Loic Poyet for the early version of this work.

References

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In *ECOOP*, volume 2374 of *LNCS*, pages 334–367. Springer Verlag, 2002.

[2] M. Anastasopoulos, J. Bayer, O. Flege, and C. Gacek. A process for product line architecture creation and evaluation – PuLSE-DSSA version 2.0. Technical Report 038.00/E, Fraunhofer IESE, 2000. URL http://publica.fraunhofer.de/documents/N-1463.html.

[3] N. Anquetil and T. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *WCRE*, pages 235–255, 1999.

[4] Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11:201–21, 1999.

[5] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in OO software. In *IWPC*, pages 153–160, 1998.

[6] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *WCRE*, pages 122–131. IEEE Press, 2004.

[7] Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *IWPC*, pages 7–17, 2001.

[8] Kent Beck and Ralph Johnson. Patterns generate architectures. In *ECOOP*, volume 821 of *LNCS*, pages 139–149. Springer-Verlag, 2004.

[9] Federico Bergenti and Agostino Poggi. Improving UML designs using automatic design pattern detection. In *SEKE*, pages 336–343, 2000.

[10] Dragan Bojic and Dusan Velasevic. A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *CSMR*, pages 23–33, 2000.

[11] I. Bowman and R. Holt. Software architecture recovery using conway's law. In *CASCON*, pages 123–133, 1998.

[12] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE*, pages 555–563, 1999.

[13] Reinder J. Bril and André Postma. An architectural connectivity metric and its support for incremental re-architecting of large legacy systems. In *IWPC*, pages 269–280, 2001.

[14] Ruven Brooks. Towards a theory of the comprehension of computer programs. *Jour. of Man-Machine Studies*, 18:543–554, 1983.

[15] Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Master's thesis, North Carolina State Univ., 1996. URL http://www.ksccary.com/kbrown.htm.

[16] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture–A System of Patterns*. Wiley & Sons, 1996. ISBN 0-471-95869-7.

[17] Ian Carmichael, Vassilios Tzerpos, and Rick C. Holt. Design maintenance: Unexpected architectural interactions. In *ICSM*, pages 134–140, 1995.

[18] Mario Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwe. A qualitative comparison of three aspect mining techniques. In *IWPC*, pages 13–22, 2005.

[19] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *ICSM*, pages 241–249, 2000.

[20] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Equipping the reflexion method with automated clustering. In *WCRE*, pages 89–98, 2005.

[21] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Jour. of Systems and Software*, 28:117–127, 1995.

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

[23] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1– The FAMOOS Information Exchange Model. Technical report, Univ. Bern, 2001.

[24] Lei Ding and Nenad Medvidovic. Focus: A light-weight, incremental approach to software architecture recovery and evolution. In *WICSA*, pages 191–201, 2001.

[25] Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS OO Reengineering Handbook*. Univ. Bern, 1999.

[26] Stéphane Ducasse and Sander Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance*, 15: 345–373, 2003.

[27] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *CSMR*, pages 309–318, 2004.

[28] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In

Tools for Software Maintenance and Reengineering, RCOST/Software Technology, pages 55–71. 2005.

[29] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *ICSM*, pages 203–212, 2006.

[30] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM*, pages 94–103, 2007.

[31] J. Dueñas, W. Lopes de Oliveira, and J. de la Puente. Architecture recovery for software evolution. In *CSMR*, pages 113–120, 1998.

[32] Alastair Dunsmore, Marc Roper, and Murray Wood. Oo inspection in the face of delocalisation. In *ICSE*, pages 467–476, 2000.

[33] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. Gupro–generic understanding of programs. *ENTCS*, 72(2), 2002.

[34] Amnon H. Eden. Visualization of OO architectures. In *ICSE workshop on Software Visualization*, 2001.

[35] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29:210–224, 2003.

[36] Wolfgang Eixelsberger, Michaela Ogris, Harald Gall, and Bernt Bellay. Software architecture recovery of a program family. In *ICSE*, pages 508–511, 1998.

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

[38] Jean-Marie Favre. Meta-model and model co-evolution within the 3D software space. In *Work. on Evolution of Large-Scale Industrial Software*, 2003.

[39] Jean-Marie Favre. CacOphoNy: Metamodel-driven software architecture reconstruction. In *WCRE*, pages 204–213, 2004.

[40] Loe Feijs and Roel De Jong. 3D visualization of software architectures. *CACM*, 41(12):73–78, 1998.

[41] Loe Feijs, René Krikhaar, and Rob van Ommering. A relational approach to support software architecture analysis. *Soft.: Practice & Exper.*, 28 (4):371–400, 1998.

[42] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[43] R. Fiutem, G. Antoniol, P. Tonella, and E. Merlo. Art: an architectural reverse engineering environment. *Journal of Software Maintenance*, 11 (5):339–364, 1999.

[44] Roberto Fiutem, Paolo Tonella, Giuliano Antoniol, and Ettore Merlo. A cliché-based environment to support architectural reverse engineering. In *ICSM*, 1996.

[45] Keith Gallagher, Andrew Hatch, and Malcolm Munro. A framework for software architecture visualization assessment. In *VISSOFT*, pages 76–81, 2005.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison Wesley, 1995.

[47] Gansner and North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000.

[48] David Garlan. Software architecture: a roadmap. In *ICSE*, pages 91–101, 2000.

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

[50] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, 2000.

[51] Jean-Francois Girard and Rainer Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *ICSM*, 1997.

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

[53] John Grundy and John Hosking. High-level static and dynamic visualisation of software architectures. In *Symposium on Visual Languages*, pages 5–12, 2000.

[54] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *TOOLS*, pages 296–305, 2001.

[55] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *WCRE*, pages 172–181, 2004.

[56] Yann-Gaël Guéhéneuc, Kim Mens, and Roel Wuyts. A comparative framework for design recovery tools. In *CSMR*, 2006.

[57] Yanbing Guo, Atlee, and Kazman. A software architecture reconstruction method. In *WICSA*, pages 15–34, 1999.

[58] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *CSMR*, pages 112–121, 2005.

[59] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A survey of trace exploration tools and techniques. In *CASON*, pages 42–55, 2004.

[60] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *ICSE*, 1995.

[61] Hassan and Holt. Using development history sticky notes to understand software architecture. In *IWPC*, pages 183–193, 2004.

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

[63] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC*, pages 94–104, 2003.

[64] Christine Hofmeister, Robert L. Nord, and Dilip Soni. *Applied Software Architecture*. Addison Wesley, 2000.

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

[66] Ric Holt. Sofware architecture as a shared mental model. In *ASERC Workshop on Software Architecture*, 2001.

[67] Richard Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE*, pages 210–219, 1998.

[68] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving OO Systems*. PhD thesis, Vrije Universiteit Brussel, 1998.

[69] Gang Huang, Hong Mei, and Fu-Qing Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering*, 13(2):257–281, 2006.

[70] IEEE. Recommended practice for architectural description for software-intensive systems. Technical report, Architecture Working Group, 2000.

[71] Ivkovic and Godfrey. Enhancing domain-specific software architecture recovery. In *IWPC*, pages 266–276, 2003.

[72] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In *WCRE*, pages 56–65, 1997.

[73] Rick Kazman and Len Bass. Categorizing business goals for software architectures. Technical report, Carnegie Mellon Univ., SEI, 2005.

[74] Rick Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Soft. Engineer.*, 1999.

[75] Rick Kazman and S. Jeromy Carriere. View extraction and view fusion in architectural understanding. In *International Conference on Software Reuse*, 1998.

[76] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *ICSE*, pages 81–90, 1994.

[77] Rick Kazman, Mark H. Klein, Mario Barbacci, Thomas A. Longstaff, Howard F. Lipson, and S. Jeromy Carrière. The architecture tradeoff analysis method. In *ICECCS*, pages 68–78, 1998.

[78] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines. Technical report, Carnegie Mellon Univ., SEI, 2001.

[79] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines, third edition. Technical report, Carnegie Mellon Univ., SEI, 2003.

[80] Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4(4640):143–162, 2007.

[81] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-Based Reverse Engineering of Design Components. In *ICSE*, pages 226–235, 1999.

[82] Jens Knodel, Isabel John, Dharmalingam Ganesan, Martin Pinzger, Fernando Usero, Jose L. Arciniegas, and Claudio Riva. Asset recovery and their incorporation into product lines. In *WCRE*, pages 120–129, 2005.

[83] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *CSMR*, pages 279–294, 2006.

[84] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[85] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in OO software. In *WCRE*, pages 208–216, 1996.

[86] Rene Krikhaar. *Software Architecture Reconstruction*. PhD thesis, Univ. Amsterdam, 1999. URL http://www.cs.vu.nl/~x/sar/sar.pdf.

[87] Philippe Kruchten, Henk Obbink, and Judith Stafford. The past, present, and future for software architecture. *IEEE Software*, 23(2):22–30, 2006.

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

[89] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*,

49(3):230–243, 2007.

[90] Danny Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA*, pages 342–357, 1995.

[91] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE*, pages 214–223, 2005.

[92] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *TSE*, 29(9):782–795, 2003.

[93] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, 1985. ISBN 0-12-442440-6.

[94] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *ENTCS*, volume 94, pages 7–18, 2004.

[95] Qingshan Li, Hua Chu, Shengming Hu, Ping Chen, and Zhao Yun. Architecture recovery and abstraction from the perspective of processes. In *WCRE*, pages 57–66, 2005.

[96] Jonas Lundberg and Welf Löwe. Architecture recovery by semi-automatic component identification. *ENTCS*, 82(5), 2003.

[97] Mircea Lungu, Adrian Kuhn, Tudor Gîrba, and Michele Lanza. Interactive exploration of semantic clusters. In *VISSOFT workshop*, pages 95–100, 2005.

[98] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *CSMR*, pages 185–196. IEEE Press, 2006. doi: 10.1109/CSMR.2006.39.

[99] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeffrey Kramer. Specifying distributed software architectures. In *ESEC*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag, 1995.

[100] Spiros Mancoridis and Brian S. Mitchell. Using automatic clustering to produce high-level system organizations of source cods. In *IWPC*. IEEE Press, 1998.

[101] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *ACM Symposium on Software Visualization*, page 27. IEEE, 2003.

[102] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *WCRE*, pages 214–223, 2004.

[103] Medvidovic and Taylor. A classification and comparison framework for software architecture description languages. *TSE*, 26(1):70–93, 2000. doi: 10.1109/32.825767.

[104] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2): 225–256, 2006. ISSN 0928-8910. doi: 10.1007/s10515-006-7737-5.

[105] Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by architectural discovery and recovery. In *International Workshop from Software Requirements to Architectures*, 2003.

[106] Nabor C. Mendonça and Jeff Kramer. Requirements for an effective architecture recovery framework. In *ISAW*, pages 101–105, 1996.

[107] Nabor C. Mendonça and Jeff Kramer. An approach for recovering distributed system architectures. *Automated Software Engineering*, 8 (3-4):311–354, 2001. ISSN 0928-8910.

[108] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[109] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views–a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006. URL http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-26.pdf.

[110] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis)*, pages 135–144. ACM Press, 2006. doi: 10.1145/1148493.1148513. URL http://www.iam.unibe.ch/~scg/Archive/Papers/Meye06aMondrian.pdf.

[111] Paul Miodonski, Thomas Forster, Jens Knodel, Mikael Lindvall, and Dirk Muthig. Evaluation of software architectures with eclipse. Technical report, Fraunhofer IESE, 2004.

[112] Brian S. Mitchell and Spiro Mancoridis. On the automatic modularization of software systems using the bunch tool. *TSE*, 32(3):193–208, 2006.

[113] Hausi A. Müller, Kenny Wong, and Scott R. Tilley. Understanding software systems using reverse engineering technology. In *OO Tech. for Database and Soft. Sys.*, pages 240–252. World Scientific, 1995.

[114] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT*, pages 18–28. ACM Press, 1995.

[115] Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, Univ. Washington, 1996.

[116] Matthias Naab. Evaluation of graphical elements and their adequacy for the visualization of software architectures. Master's thesis, Fraunhofer IESE, 2005.

[117] Jörg Niere, Jörg P. Wadsack, and Lothar Wendehals. Design pattern recovery based on source code analysis with fuzzy logic. tr-ri-01-222, Software Engineering Group, Univ. Paderborn, Germany, 2001.

[118] Bounour Nora, Ghoul Said, and Atil Fadila. A comparative classification of aspect mining approaches. *Journal of Computer Science*, 4(2):322–325, 2006. ISSN 1549-3636.

[119] Liam O'Brien, Christoph Stoermer, and Chris Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical Report CMU/SEI-2002-TR-024, Carnegie Mellon Univ., 2002.

[120] Liam O'Brien, Dennis Smith, and Grace Lewis. Supporting migration to services using software architecture reconstruction. In *International Workshop on Software Technology and Engineering Practice*, pages 81–91. IEEE Press, 2005. ISBN 076952639X. doi: 10.1109/STEP.2005.29. URL http://portal.acm.org/citation.cfm?id=1158338.1158738.

[121] Michael Pacione. *A Novel Software Visualisation Model to Support OO Program Comprehension*. PhD thesis, Univ. Strathclyde, 2005.

[122] Thomas Panas. *A Framework for Reverse Engineering*. PhD thesis, Vajjo Univ., 2005.

[123] Ilian Pashov and Matthias Riebisch. Using feature modelling for program comprehension and software architecture recovery. In *Engineering of Computer-Based Systems (ECBS)*, 2004.

[124] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17 (4):40–52, 1992. URL http://www.bell-labs.com/user/dep/work/papers/swa-sen.ps.

[125] Ilka Philippow, Detlef Streitferdt, and Matthias Riebisch. Design pattern recovery in architectures for supporting product line development and application. In *ECOOP workshop–Modeling Variability for OO Product Lines*, pages 42–57, 2003.

[126] Martin Pinzger. *ArchView–Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna Univ. of Technology, 2005.

[127] Martin Pinzger and Harald Gall. Pattern-supported architecture recovery. In *IWPC*, pages 53–61, 2002. doi: 10.1109/WPC.2002.1021318.

[128] Martin Pinzger, Michael Fischer, Harald Gall, and Mehdi Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *WCRE*, pages 170–178, 2002. doi: 10.1109/WCRE.2002.1173075.

[129] Martin Pinzger, Harald Gall, Jean-Francois Girard, Jens Knodel, Claudio Riva, Wim Pasman, Chris Broerse, and Jan Gerben Wijnstra. Architecture recovery for product families. In *International Workshop on Product Family Engineering (PFE-5)*, volume 3014 of *LNCS*, pages 332–351. Springer-Verlag, 2004. URL http://www.infosys.tuwien.ac.at/Cafe/doc/mp-ar_for_families.pdf.

[130] Martin Pinzger, Harald Gall, and Michael Fischer. Towards an integrated view on architecture and its evolution. *ENTCS*, 127(3):183–196, 2005.

[131] Trygve Reenskaug. *Working with Objects: The OOram S.E. Method*. Manning Publications, 1996. ISBN 1-884777-10-4.

[132] Tamar Richner and Stéphane Ducasse. Recovering high-level views of oo applications from static and dynamic information. In *ICSM*, pages 13–22, 1999. doi: 10.1109/ICSM.1999.792487.

[133] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *ICSM*, page 34, 2002. doi: 10.1109/ICSM.2002.1167745.

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

[135] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. In *CSMR*, page 47. IEEE Press, 2002.

[136] Jason E. Robbins and David F. Redmiles. Software architecture critics in the argo design environment. *Knowledge-Based Systems*, 11(1): 47–60, 1998. doi: 10.1016/S0950-7051(98)00055-0.

[137] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *ICSM*, pages 72–81. IEEE Press, 2004. doi: 10.1109/ICSM.2004.1357792.

[138] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *OOPSLA*, pages 167–176, 2005.

[139] Kamran Sartipi. *Software Architecture Recovery based on Pattern Matching*. PhD thesis, School of Computer Science, Univ. Waterloo, Canada, 2003.

[140] Mohlalefi Sefika. *Design Conformance Management of Software Systems: an Architecture-Oriented Approach*. PhD thesis, Univ. Illinois, 1996.

[141] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996. ISBN 0-13-182957-2.

[142] Forrest Shull, Walcelio L. Melo, and Victor R. Basili. An inductive method for discovering design patterns from OO soft. sys. Technical Report CS-TR-3597, Univ. Maryland, 1996.

[143] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *TSE*, 25(6):749–768, 1999.

[144] Susan Elliott Sim, Charles L.A. Clarke, Richard C. Holt, and Anthony M. Cox. Browsing and searching software architectures. In *ICSM*, page 381. IEEE Press, 1999. doi: 10.1109/ICSM.1999.792636.

[145] Daniel Simon and Thomas Eisenbarth. Evolutionary introduction of software product lines. In *SPLC*, pages 272–282. Springer-Verlag, 2002. ISBN 3-540-43985-4.

[146] Smolander, Hoikka, Isokallio, Kataikko, Mäkelä, and Kälviäinen. Required and optional viewpoints–what is included in software architecture? Technical report, Univ. Lappeenranta, 2001.

[147] Periklis Sochos, Matthias Riebisch, and Ilka Philippow. The feature-architecture mapping (FArM) method for feature-oriented development of software product lines. In *ECBS*, pages 308–318. IEEE Press, 2006.

[148] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *ICSE*, pages 196–207. ACM Press, 1995.

[149] Christoph Stoermer and Liam O'Brien. Map–Mining architectures for product line evaluations. In *WICSA*, pages 35–41, 2001. ISBN 0-7695-1360-3.

[150] Christoph Stoermer, Liam O'Brien, and Chris Verhoef. Moving towards quality attribute driven software architecture reconstruction. In *WCRE*, pages 46–56. IEEE Press, 2003. doi: 10.1109/WCRE.2003.1287236.

[151] Christoph Stoermer, Anthony Rowe, Liam O'Brien, and Chris Verhoef. Model-centric software architecture reconstruction. *Software: Practice and Exper.*, 36(4):333–363, 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:4.

[152] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP Views. In *ICSM*, pages 275–284. IEEE Press, 1995.

[153] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44: 171–185, 1999.

[154] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE*, 2001.

[155] Davor Svetinovic and Michael Godfrey. A lightweight architecture recovery process. In *WCRE*, 2001.

[156] Tarja Systä. On the relationships between static and dynamic models in reverse engineering java software. In *WCRE*, pages 304–313, 1999.

[157] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba–an environment for reverse engineering Java software systems. *Software: Practice and Experience*, 31(4):371–394, 2001. ISSN 0038-0644. doi: 10.1002/spe.386.

[158] Telea, Maccari, and Riva. An open visualization toolkit for reverse architecting. In *IWPC*, pages 3–13. IEEE Press, 2002. doi: 10.1109/WPC.2002.1021303.

[159] Scott R. Tilley, Dennis B. Smith, and Santanu Paul. Towards a framework for program understanding. In *IWPC*, page 19. IEEE Press, 1996. ISBN 0-8186-7283-8. doi: 10.1109/WPC.1996.501117.

[160] Thomas Tilley, Richard Cole, Peter Becker, and Peter Eklund. A survey of formal concept analysis support for software engineering activities. In *ICFCA*. Springer-Verlag, 2003.

[161] Paolo Tonella. Concept analysis for module restructuring. *TSE*, 27(4): 351–363, 2001.

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

[163] Adrian Trifu. *Using Cluster Analysis in the Architecture Recovery of OO Systems*. PhD thesis, Univ. Karlsruhe, 2001.

[164] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. In *ICSM*, pages 398–407, 2001.

[165] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. In *WICSA*, pages 122–134, 2004.

[166] Aline Vasconcelos and Cláudia Werner. Software architecture recovery based on dynamic analysis. In *Brazilian Symp. on Softw. Engineering*, 2004.

[167] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient mapping of software system traces to architectural views. In *CASCON*, page 12. IBM Press, 2000.

[168] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In *ICSE Workshop on Dynamic Analysis (WODA)*, 2003.

[169] Theo Wiggerts. Using clustering algorithms in legacy systems remodularization. In *WCRE*, pages 33–43. IEEE Press, 1997.

[170] Norman Wilde and Ross Huitt. Maintenance support for OO programs. *TSE*, SE-18(12):1038–1044, 1992.

[171] Norman Wilde and Michael Scully. Software reconnaisance: Mapping program features to code. *Journal on Software Maintenance*, 7(1): 49–62, 1995.

[172] Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003. ISSN 0164-1212. doi: 10.1016/S0164-1212(02)00052-3.

[173] Kenny Wong. The rigi user's manual–version 5.4.4. Technical report, Univ. Victoria, 1998.

[174] W. Eric Wong, Joseph R. Horgan, Swapna S. Gokhale, and Kishor S. Trivedi. Locating program features using execution slices. *Asset*, 00: 194, 1999. doi: 10.1109/ASSET.1999.756769.

[175] Steven G. Woods, S. Jeromy Carrière, and Rick Kazman. The perils and joys of reconstructing architectures. *SEI Interactive, The Architect*, 2, 1999.

[176] Lei Wu, Houari Sahraoui, and Petko Valtchev. Program comprehension with dynamic recovery of code collaboration patterns and roles. In *CASCON*, pages 56–67. IBM Press, 2004.

[177] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *WCRE*, pages 90–99. IEEE Press, 2004.

[178] Roel Wuyts. Declarative reasoning about the structure oo systems. In *TOOLS USA*, pages 112–124. IEEE Press, 1998.

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

[180] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. DiscoTect: A system for discovering architectures from running systems. In *ICSE*, pages 470–479, 2004.

[181] A.S. Yeh, D.R. Harris, and M.P. Chase. Manipulating recovered software architecture views. In *ICSE*, 1997.