Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Decentralized Software Architecture Discovery in Distributed Systems

Jason Porter jporte10@gmu.edu
Daniel A. Menascé menasce@gmu.edu
Hassan Gomaa hgomaa@gmu.edu

Technical Report GMU-CS-TR-2016-2

Abstract

Software architecture discovery plays an increasingly important role in the evolution, maintenance, and runtime self-adaptation of modern software systems whose architecture may have become outdated or may not have previously existed. However, current approaches to architecture discovery take a centralized approach, in which the process is carried out from a single location. This proves inadequate in the case of large distributed systems which, due to size, consist of nodes that are disparately located and are highly dynamic in nature. This report presents DeSARM: **De**centralized **S**oftware **A**rchitecture discove**R**y **M**echanism, a completely decentralized and automated approach for software architecture discovery of distributed systems based on gossiping and message tracing. Through message tracing, the technique is able to identify important architectural characteristics such as components and connectors, in addition to synchronous and asynchronous communication patterns. Furthermore, through its use of gossiping, it exhibits the properties of scalability, global consistency among participating nodes, self-organization, and resiliency to failures. The report discusses DeSARM's architecture and detailed design and demonstrates its properties through an analysis of small and large-scale experiments.

1 Introduction

Software architecture—the high level structures of a software system including a collection of components, connectors and constraints—plays an increasingly critical role in the design and development of any large complex software system. These artifacts are needed to reason about the system. In general, software architecture acts as a bridge between requirements and implementation and provides a blueprint for system construction and composition. The architecture helps in the understanding of complex systems, supports reuse at both the component and architectural level, indicates the major components to be developed and their relationships and constraints, exposes changeability of the system as well as allowing the verification and validation of the target system at a high level [22, 56].

An area in which software architecture has been very influential is that of self-adaptive systems, i.e., systems that are capable of self-configuration, self-optimization, self-healing and self-protection, also called self-* or autonomic systems [25]. In architecture-based self-adaptation, components dynamically change in order to continuously adhere to architectural specifications and system goals. This approach has proven to be very popular and many groups have used it as the foundation for their work [59]. In a previous work [37], Menascé et al. developed a framework for a self-architecting software system (SASSY), which was designed to automate the architectural decision making process for service-oriented systems in the face of quality-of-service (QoS) trade-offs. The framework automatically generates, at runtime, candidate software architectures and selects the one that best serves stakeholder-defined scenario-based QoS goals [14, 35]. Adaptation decisions are made based on changes in the system's operational environment that affect these goals [19, 18]. A new architecture is then generated and the system is reconfigured to a new QoS optimized state.

Whereas with SASSY the current (i.e., before adaptation) architecture is assumed to be known, this research considers the case in which the architecture is unknown and needs to be discovered at run-time. It is not uncommon for the architecture to be unknown in large-scale distributed systems because the system's structure is often dynamic due to churn, where nodes randomly join and leave the network and may fail.

Software architecture erosion may occur when the prescriptive or intended software architecture departs

from the descriptive or implemented software architecture due to the system's evolution over time [10]. As one can imagine, such discrepancies can affect adaptation decisions at run-time where complete knowledge of the current architecture is imperative. Besides erosion, another factor that would require the retrieval of the software architecture at run-time is when there is no prescriptive architecture available for the system. This often occurs when the system is either developed without an explicit architecture or design documents may be lost [22]. Such issues are a motivation for software architecture discovery. Software architecture discovery involves the methods, techniques and processes used to uncover a software system's architecture from available information [38]. This is the focus of this report.

Currently, most architecture discovery techniques rely on input obtained from implementation level artifacts to reconstruct the software architecture [39]. Software maintenance and evolution requires architecture discovery when the original architecture has eroded [16]. In our work we do architecture discovery for architecture-based adaptation purposes and perform discovery through a decentralized analysis of message flows between components in a distributed system.

A software architecture has structure and behavior, where structure captures components and how they are connected, and behavior captures interactions among components. This report focuses on software architecture structure discovery but also addresses behavioral discovery of the architecture such as inter-component communication patterns (i.e., synchronous, asynchronous, single or multiple destination). We recover the software architecture in a decentralized manner by keeping message logs in each node and disseminating message interaction information between nodes through the use of *gossip* exchanges. Once convergence is achieved, each component will have a global view of the architecture.

This report makes the following key contributions:

1. DeSARM: **De**centralized **S**oftware **A**rchitecture discove**R**y **M**echanism, a completely decentralized and automated method for software architecture discovery of distributed systems using gossiping and message tracing. The information dissemination and fast convergence capability of gossiping aids each component in deriving a view of the architecture.

2. Demonstration that the software architecture discovery mechanism exhibits the following properties: self-healing, self-organizing, global consistency, and scalability. These properties relate to the architecture discovery process not to the failure recovery of the application system.

The rest of this report is organized as follows. Section 2 discusses related work. Section 3 provides a problem description along with some basic assumptions. Section 4 describes the architecture of DeSARM, our architecture discovery method. Section 5 presents the details of running experiments with DeSARM. Finally, Section 6 presents some concluding remarks and discusses possible future work.

2 Related Work

The columns of Table 1 categorize software architecture discovery techniques into either centralized or distributed and the rows categorize software applications as either centralized or distributed. A centralized architecture discovery method is one in which the discovery process, which includes data gathering and processing, is done from a single location. Conversely, in a distributed discovery method, data gathering and processing is accomplished across multiple locations. We discuss here prior work that uses centralized discovery methods applied to both centralized and distributed applications, as well as distributed discovery methods applied to distributed applications, which is the focus of our report. Clearly, a distributed method applied to a centralized application is not applicable.

	Architecture Discovery Method	
Application	Centralized	Distributed
Centralized	✓	N/A
Distributed	✓	✓

Table 1: application structure and architecture discovery method

Software architecture discovery approaches can be further classified into dynamic, static, and hybrid, i.e., combining both dynamic and static analysis, as described in what follows.

Israr et al. [23] describe SAMEtech, a dynamic approach for automating the discovery of architecture models and layered performance models from message trace information. DiscoTect [53] uses a set of pattern recognizers and knowledge of the architectural style being implemented to map low-level system events into high-level architecturally meaningful events. Bojic and Velasevic [6] use test cases that cover relevant use-cases, and concept analysis to group system entities together that implement similar functionality. Vasconcelos et al. [58] use specified use-cases to generate execution traces from which interaction patterns are identified using pattern detection in order to define architectural elements. Esfahani et al [13] take a dynamic approach to recovering the architectural model of a distributed application by generating a component interaction model using data mining. The approach works by collecting system execution traces at run-time, then uses association rule mining to infer a probabilistic model of the component interactions taking place.

A number of papers present hybrid approaches for architecture discovery. For instance, in [4] Antoniol et. al propose WANDA, a tool for instrumenting web applications that combines the static analysis of server-side scripting with the dynamic analysis of HTML, SQL and client-side scripting to recover the as-is architecture. Quingshan et al. [46] deals with architecture discovery from the perspective of processes, by extracting Process Structure Graphs (PSGs) based on the features of the relations among processes in a Unix system. In their technique, static analysis is used to identify the static code fragment of a dynamic process, then during dynamic analysis a mapping algorithm is used to identify the correspondence between the dynamic process ID and the static process module. Riva and Yang [48] present an approach to create an architectural model of a system along with its documentation using XML. Their technique relies on static analysis of the source code and system documentation along with execution traces gained through dynamic analysis. Riva and Rodriguez [47] and Sartipi and Dezhkam [50] combine static and dynamic information to reconstruct an architecture with multiple views. In the former case, the static information is extracted by analyzing the source code and searching for architecturally significant elements while the dynamic information is obtained through instrumenting the source code and executing different scenarios, then collecting architecturally relevant traces. In the latter case, the static information is generated using an approximate pattern matching technique to generate a source graph, while the dynamic information is obtained by observing the number of function invocations for each function that is involved in the execution of frequent task scenarios. The obtained dynamic information is then embedded into the extracted source graph of the system to be used for an amalgamated static and dynamic discovery process.

A large number of architecture discovery approaches rely on purely static analysis. There are also many different types of static analyses to gather architectural knowledge. Moreover, source code is not the only artifact that can be analyzed statically; other examples include build files, scripts, configuration files and natural language text [27]. Some approaches directly query the source code [41] [43] [44] while others abstract it and represent it as metamodels [32] [31] [21]. Some approaches use textual information derived from the source code and comments [8] [7] [40] [17] [43] [44] as well as method names [29] [34] and source file names [3]. Another approach is based on the physical organization of files, folders and packages within an application [20] [62] [32] [30] [45] [61]. Clustering can also be used to group these implementation-level artifacts into clusters representing components [27] [33] [60]. Pattern detection [57] [51] [52] and conceptual analysis [54] is used to identify structural dependencies between components in some approaches. Another set of approaches use hierarchical clustering to recover architectures that can be viewed at multiple levels of abstraction [2] [33] [42]. In [38] Mendonça describes X-ray, an integrated approach for statically recovering distributed system architectural views. X-ray combines component module classification which automatically distinguishes source code modules according to the executable components they implement, syntactic pattern matching which recognizes code fragments that implement typical component interaction features, and structural reachability analysis which associates those features to the code specific to each component.

All of the approaches mentioned above are based on a centralized method to architecture discovery applied to both centralized and distributed systems. In contrast, our approach is based on a decentralized method to architecture discovery. The underlying assumption in the current approaches is that the gathering and processing of system events can take place centrally. However, in the case of a large-scale distributed system such a premise proves infeasible. To the best of our knowledge, no existing method in the current literature takes a completely decentralized approach to the software architecture discovery process.

Another area related to our work is that of message logging in distributed systems. Message logging protocols are more often used for achieving fault tolerance and are an integral part of implementing processes that can recover from failures [24] [55] [1]. Our approach is dependent on access to and availability of messaging events for architecture discovery, also due to the expected size and dynamicity of our target system, failures are considered imminent. As such, employing such protocols is deemed appropriate.

3 Problem Description

A distributed software system consists of a number of software components running on two or more interconnected nodes. More than one component may be running at any given node.

This report makes the following assumptions:

1. Nodes, links, and components may fail in either a fail-recover mode or in a fail-stop mode.
2. If a component fails, it is restarted on the same node if the node is still running.
3. If a node fails, all the components running on that node fail.
4. If a node cannot be restarted, its components can be moved to other node(s) using an existing component recovery mechanism not within the scope of our work.
5. The message exchange between components can use either a connection-less transport protocol such as UDP or a connection-oriented protocol such as TCP.

CENTRALIZED	
Dynamic Analysis	
Filtering	Schmerl (2006)
Pattern Matching	Schmerl (2006) Israr (2007) Vasconcelos (2004)
Clustering	Bojic (2000) Israr (2007)
Concept Analysis	Bojic (2000)
Tracing	Israr (2007) Esfahani (2016)
Data Mining	Esfahani (2016) Yuan (2016)
Hybrid	
Filtering	Antoniol (2004) Riva (2002) Sartipi (2007)
Pattern Matching	Riva (2002) Sartipi (2007)
Clustering	Riva (2002) Sartipi (2007) Riva (2002)
Concept Analysis	Sartipi (2007)
Standard Visualization	Antoniol (2004) Qingshan (2005) Riva (2002) Riva (2002) Sartipi (2007)
Slicing	Qingshan (2005) Riva (2002)
Querying	Sartipi (2007)
Metrics	Sartipi (2007)
Static Analysis	
Clustering	Koschke (2009) Maqbool (2007) Wiggerts (1997) Andritsos (2005) Naseem (2011)
Pattern Matching	Tzerpos (2000) Sartipi (2001) Sartipi (2003) Mendonca (2011)
Concept Analysis	Siff (1999)
Querying	Murphy (1995) Pinzger (2002) Pinzger (2002)
Meta-models	Lungu (2006) Lethbridge (2004) Schurr (2006)
Text Analysis	Pinzger (2002) Pinzger (2002) Corazza (2010) Corazza (2011) Misra (2012) Garcia (2011) Kuhn (2007) Marcus (2004) Anquetil (1999)
Physical Organization	Harris (1995) Yeh (1997) Langelier (2005) Pinzger (2005) Wu (2004)
Reachability	Mendonca (2011)
Dominance	Mendonca (2011)
DECENTRALIZED	
Dynamic Analysis	
Message Tracing	Our Approach
Gossiping	Our Approach

Table 2: Software Architecture Discovery Approaches

6. The message exchange between DeSARM modules should use TCP if the messages need to be broken into more than one packet.
7. The software architecture is not known because it may dynamically change due to churn and failures or it may not have previously existed.

These important assumptions are key to the foundations of our architecture discovery process. We believe each assumption reasonably reflects important considerations related to the dynamic execution of a large-scale distributed component-based software system.

4 Software Architecture Discovery Method

This section provides a detailed discussion of our architecture discovery method, DeSARM. We first describe the structure of a node running DeSARM. Then, we give an overview of how gossiping and message tracing are incorporated into the discovery process. Finally, we delve into the details of the architecture discovery method.

A node in a distributed system consists of three layers according to Fig. 1: application layer, DeSARM layer, and communication middleware. The application layer consists of the distributed application components which communicate over the network via messaging events. Each component has two logs: message sent log (MSL) and message received log (MRL). The DeSARM layer forms a wrapper around the communication middleware and provides an interface to the application layer components. DeSARM consists of a number of modules each providing different functionality:

- Message logging: All incoming messages are logged before being passed to the application layer components and all outgoing messages are logged before being passed to the communication middleware. Also, in compliance with message logging protocols, all messages are logged to stable storage.
- Message log aggregation: The MSLs and MRLs of all components are aggregated to form an aggregate message log (AML) at each node. This is further merged with the AMLs from incoming gossip messages received from other nodes (see below).
- Gossip-based dissemination: This forms the core of our architecture discovery method and enables the distribution of AMLs from each node throughout the system.
- Peer node selection: This is achieved through the maintenance of a component/node database which is derived by identifying component IDs and their related node IDs from incoming and outgoing messages. This ensures that only nodes running components that are part of the same application are selected for dissemination.

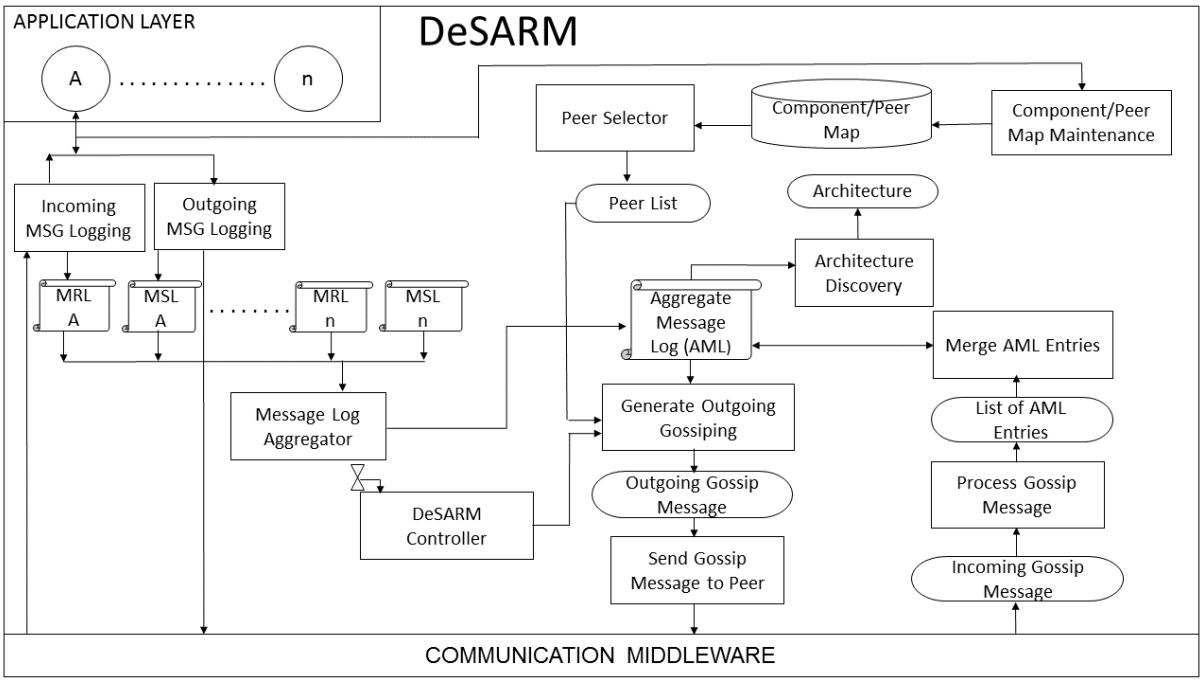

Figure 1: Node Structure

- Control: This manages the execution of the gossip process by maintaining the timing between consecutive rounds.
- Architecture discovery: This is used to derive the architectural view of the system based on the message traces.

Finally, the communication middleware provides network access allowing the sending and receiving of component-level and gossiping messages between nodes.

Gossiping is an epidemic protocol, which due to its simplicity, robustness and flexibility makes it ideal for reliable dissemination of data in large-scale distributed systems. An important observation of gossip-based dissemination is that data spreads exponentially fast and takes $O(\log N)$ rounds to reach all nodes, where N is the number of nodes in the system [26]. The essence of this approach, which lies at the core of all gossip-based dissemination approaches, was first introduced in the seminal paper by Demers et al. [9], and involves the dissemination of data by allowing randomly chosen pairs of nodes to exchange new information. After the exchange, the two nodes forming a pair should have the same information effectively reducing the entropy of the system [49].

The main elements of the gossip-based dissemination framework are: *peer selection*, where a peer (node) selects another peer uniformly at random from the set of available peers, *data exchanged*, which involves the exchange of data between peers and is specific to the use of the gossip mechanism, and *data processing*, which details how

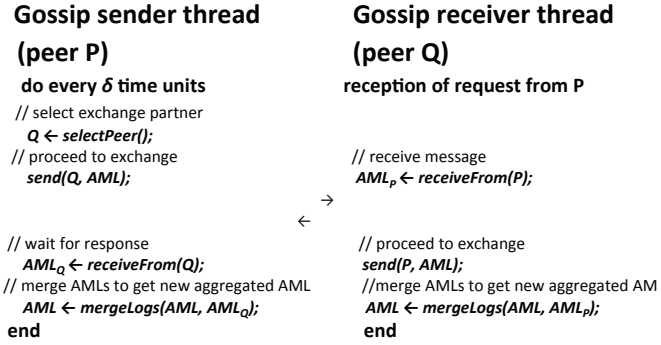

Figure 2: Gossip-based Dissemination Framework

each peer handles the information received from other peers and is also specific to the use of the gossip mechanism [26]. DeSARM's use of gossip-based dissemination is depicted in Fig. 2 and is described as:

- **Peer selection:** A peer P periodically chooses another peer Q uniformly at random from the set of available peers.
- **Data exchanged:** The AML is copied from one peer to another.
- **Data processing:** The received AML is merged with the local AML at each node to produce an updated AML.

Once the gossip protocol has converged, i.e., all nodes of the system have the same AML, a view of the system architecture can be derived at each node.

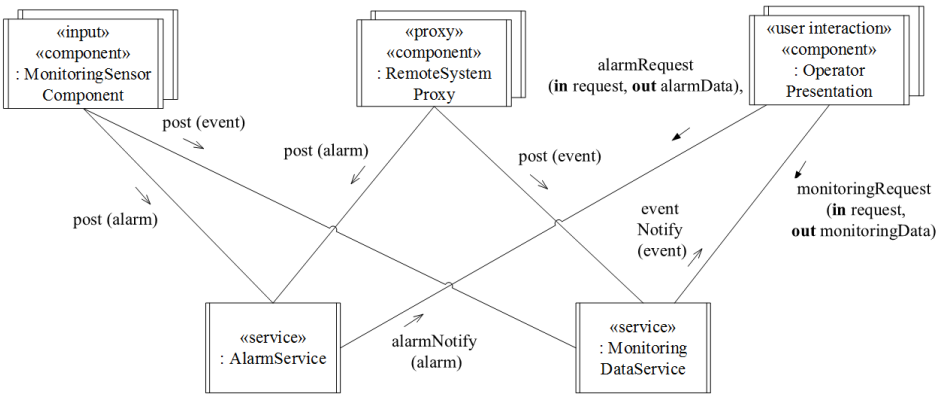

Figure 3: Emergency Monitoring System. Components: Monitoring Sensor (MSC, MSC2, and MSC3), Remote System Proxy (RSP, RSP2, and RSP3), Operator Presentation (OP), Alarm Service (ArmS), and Monitoring DataService (MDS).

To facilitate message tracing, the MSL and MRL of each component is used. Each log entry has the following fields:

- Timestamp (`ts`)
- Destination Type (`dt`): single destination (`SD`) or multiple destination (`MD`)
- Message Type (`mt`): request reply (`RQ`), no-reply requested (`NR`) and reply to previous request (`RP`)
- Transaction ID (`tid`)
- Message Unique ID (`mid`)
- Return ID (`rid`): equals 0 if `mt` is not equal to `RP`, otherwise equals `mid` of original request message
- Source Component (`src_comp`): component sending the message
- Destination Component (`dst_comp`): component receiving the message
- Node ID (`node_id`): ID of sending node

During the aggregation of the MRLs and MSLs to form the AML, only messages sent and received within a specified time interval are considered. This interval should be long enough to capture component communications that are intermittent but short enough to ensure that only the most up to date interactions are used during the discovery process.

The MRL and MSL for each component are scanned to recover messages with a timestamp `ts` within the specified time interval. The interaction patterns for these messages are then identified. The following types of messages are considered:

- Reply requests (`RQ`)
- No-reply requests (`NR`)
- Replies (`RP`).

These message types allows us to identify synchronous vs asynchronous interactions. In the former case, since reply messages are guaranteed to have a request, then the original request reply message and its associated reply message are treated as a single synchronous interaction (`SY`). If the original message was sent as a unicast then the tuple (source, destination, SY, SD) is added to the AML. Otherwise, if the original message was sent as a multicast, then the tuple (source, destination, SY, MD) is added to the AML. No-reply requested messages on the other hand are treated as asynchronous interactions (`AS`) and added to the AML as (source, destination, AS, SD) if the message was sent as a unicast, or (source, destination, AS, MD) if the message was multicasted (see Algorithm 1). The AML is treated as a set so only unique tuple entries are allowed for each component interaction, irrespective of the frequency of such interactions in the message logs. This is the case because a software architecture does not consider how many times a certain type of interaction occurred between components.

After each round of gossiping, the updated AML is used to incrementally recover the architecture, which is represented as a labeled directed graph. The vertices of this graph correspond to unique component ids and the edges correspond to unique component interactions. Edges are labeled with the interaction patterns (`SY or AS`) and destination types (`SD or MD`) (see Algorithm 2).

To depict how DeSARM works, we use an example architecture of a distributed emergency monitoring system (see Fig.3). The architecture consists of five types of components with three instances of the Monitoring Sensor and RemoteSystem Proxy components, two instances of

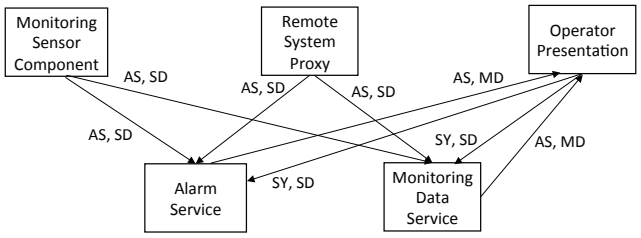

Figure 4: Recovered Architecture as Labeled Directed Graph

the Operator Presentation component and a single instance of the other components. This example assumes that each component is assigned to a single node. The communication patterns within the system are:

- Operator Presentation sends synchronous messages with reply to Alarm Service and Monitoring Data Service.
- Alarm Service and Monitoring Data Service send asynchronous multicast messages to Operator Presentation.
- Monitoring Sensor and Remote System Proxy send asynchronous unicast messages to Alarm Service and Monitoring Data Service.

The recovered architecture corresponding to Fig. 3 is the graph shown in Fig.4.

5 DeSARM Implementation and Experiments

Our experiments demonstrate the operation of DeSARM and assess its convergence and the number of messages exchanged by the DeSARM middleware. Two types of experiments were conducted. In the first, there were no component/node failures during the experiment. In the second, we added random fail-recover failures for each of the components. This second experiment reveals the impact of failures on the convergence of DeSARM to the final architecture.

We implemented DeSARM in Java and emulated a distributed system by implementing each node of the distributed system on a different virtual machine (VM). We spread the VMs over physical machines connected over a network. The VMs communicate over TCP/IP so they can be located anywhere on the network. The DeSARM implementation is heavily multi-threaded with different functions of DeSARM implemented as different threads. Some examples of threads include sending and receiving of gossip messages, message log aggregation, architecture discovery, component/node database maintenance, and sending and receiving of component

Algorithm 1: Message Log Aggregation

Input : MRL and MSL of each component
Output: AML

```
1  Definitions: C: set of components; T: time interval
2  MRL: message received log; MSL: message sent log
3  AML: aggregate message log
4  AML ← {}
5  foreach c ∈ C do
6      foreach m ∈ c.MSL do
7          if m.ts ≤ T then
8              if m.rt = RP AND ∃ m' ∈ MRL s.t. m'.mid
                = m.rid then
9                  if m'.dt = MD then
10                     AML ← AML ⋃ (m.dst_comp, c,
                         SY, MD)
11                 end
12                 else
13                     AML ← AML ⋃ (m.dst_comp, c,
                         SY, SD)
14                 end
15             end
16             else if m.rt = NR then
17                 if m.dt = MD then
18                     AML ← AML ⋃ (c, m.dst_comp,
                         AS, MD)
19                 end
20                 else
21                     AML ← AML ⋃ (c, m.dst_comp,
                         AS, SD)
22                 end
23             end
24         end
25     end
26     foreach m ∈ c.MRL do
27         if m.ts ≤ T then
28             if m.rt = RP AND ∃ m' ∈ MSL s.t. m'.mid
                = m.rid then
29                 if m'.dt = MD then
30                     AML ← AML ⋃ (c, m.src_comp,
                         SY, MD)
31                 end
32                 else
33                     AML ← AML ⋃ (c, m.src_comp,
                         SY, SD)
34                 end
35             end
36             else if m.rt = NR then
37                 if m.dt = MD then
38                     AML ← AML ⋃ (m.src_comp, c,
                         AS, MD)
39                 end
40                 else
41                     AML ← AML ⋃ (m.src_comp, c,
                         AS, SD)
42                 end
43             end
44         end
45     end
46 end
```

Algorithm 2: Architecture Discovery

Input : AML
Output: Architecture (A)

```
A: {V, E, L} /* architecture as a graph */
L ← {SY, AS, SD, MD} /* set of labels */
V ← {} /* set of vertices */
E ← {} /* set of edges */
foreach (c_i, c_j, pattern, dt) ∈ AML do
    V ← V ⋃ {c_i, c_j}
    E ← E ⋃ {(c_i, c_j, pattern, dt)}
end
```

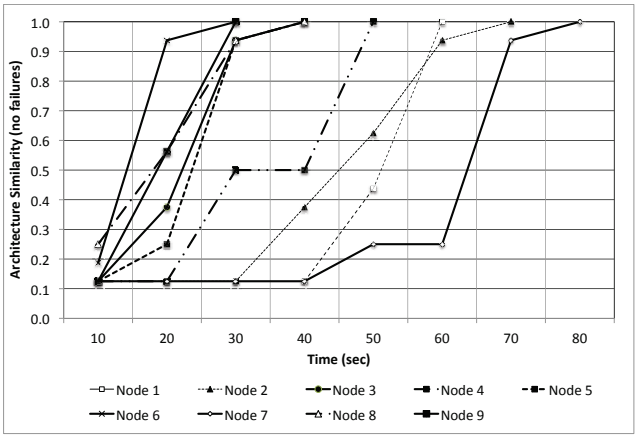

Figure 5: Architecture similarity at the 9 nodes as a function of time with no failures.

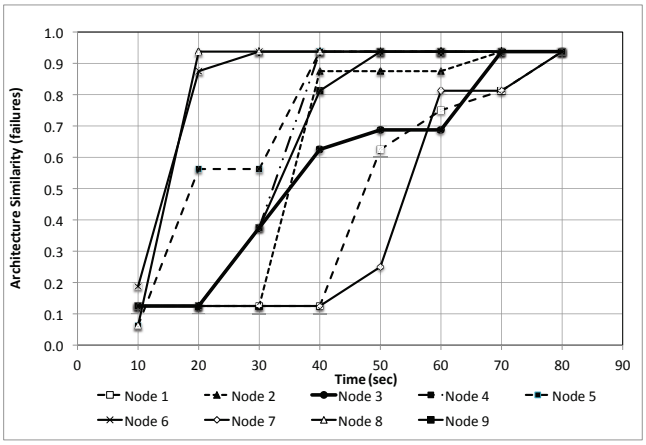

Figure 6: Architecture similarity at the 9 nodes as a function of time with component failures.

messages. All the communication between nodes uses Java sockets.

We use the application described in Fig. 3, whose architecture is known, and show that DeSARM converges exactly to that known architecture. Table 3 shows the mapping between nodes, components, and physical machines for this architecture. As discussed above, the known and recovered architectures are represented as graphs; we compare the similarity between the two (the known and current version of the recovered architecture) over time. For that, we use the graph comparison algorithms proposed in [28][64][15] and a graph similarity metric that ranges from 0 to 1, where 0 indicates no similarity and 1 indicates that the two graphs are identical. We plot the evolution of the similarity metric over time to display the convergence speed of the discovery mechanism.

Figure 5 shows how the architecture converges over time to the known architecture at each of the nodes shown in Table 3 under a no-failure case. Different nodes converge at different rates but at time 80 sec all have converged to the correct software architecture. Nodes 6 and 9 are the first to converge and node 7 is the last. Note that in our implementation of gossiping, each time a node i sends a gossip message to node j, node j replies with a gossip message. This way, two components will exchange AMLs more often, leading to faster convergence. Because of the random nature of peer selection in the gossip protocol, some components may gossip more often with interconnected components, leading to different convergence rates among nodes.

Figure 6 shows the evolution of the architecture similarity in the first 80 seconds of the experiments when component failures start to occur. As the figure shows, the architecture does not converge within that interval even though all components come close to that: all nodes have a similarity metric equal to 0.9375 (i.e., < 1) at $t = 80$ sec. The failure probability of each component, while processing, is set at 20% (a relatively high value) and the average component down time is set at 180 seconds. Thus, at approximately $t = 260$ sec, the failed components will start to recover from the failure and will continue to send messages. DeSARM automatically resumes its message collection and gossiping of newly updated AMLs when components start to recover. When that happens, convergence is achieved as illustrated in Table 4, which shows the instants at which nodes 1-9 converge after failure recovery. These instants are spread between $t = 340$ sec and $t = 400$ sec.

Table 5 shows the number of gossip messages sent and received per node until convergence is achieved in the case failures do not occur and in the case failures occur. As the table shows, the number of sent and received gossip messages when no failures occur is about 1/3 of the corresponding number when there are failures.

For illustration and debugging purposes, each node collected an event log (not part of DeSARM) during the experiments. Entries in these logs are timestamped in nanoseconds and correspond to events such as sending application-level messages, sending/receiving DeSARM gossip messages, and computing architecture similarity metrics. At the end of the experiment, the event

Table 3: Mapping of nodes to components and physical machines.

Node	Software Component	Machine
Node 1	Monitoring Sensor Component (MSC)	Machine 1
Node 2	Monitoring Sensor Component 2 (MSC2)	Machine 1
Node 3	Monitoring Sensor Component 3 (MSC3)	Machine 1
Node 4	Remote System Proxy (RSP)	Machine 1
Node 5	Operator Presentation (OP)	Machine 2
Node 6	Alarm Service (ArmS)	Machine 2
Node 7	Remote System Proxy 2 (RSP2)	Machine 1
Node 8	Monitoring Data Service (MDS)	Machine 2
Node 9	Remote System Proxy 3 (RSP3)	Machine 2

Table 4: Convergence instants of nodes 1-9 when components fail.

1	2	3	4	5	6	7	8	9
390	380	390	360	340	340	400	340	380

Table 5: Number of gossip messages sent per node.

1	2	3	4	5	6	7	8	9
Messages Sent - No failures								
24	22	20	22	21	22	25	22	21
Messages Received - No failures								
14	15	22	14	19	21	11	32	21
Messages Sent - Failures								
61	59	63	60	39	60	65	50	63
Messages Received - Failures								
36	52	50	55	54	68	69	59	59

logs of all nodes were sort-merged offline to produce a single log. Figure 7 shows a few excerpts of this log. The first two entries of this log show application-level messages sent by component MSC at Node 1 to components ArmS (at Node 6) and MDS (at Node 8). The next two entries correspond to similar messages sent by MSC2 at Node 2, to components ArmS and MDS. Later in time, DeSARM at Node 1 sends a gossip message to Node 8 with Node 1's current view of the AML, namely [(MSC,ArmS,AS,SD), (MSC,MDS,AS,SD)]. This view only reflects the messages that component MSC at Node 1 sent to nodes 6 and 8. Later in time, DeSARM at Node 8 receives the following gossip message from Node 4: [(RSP,ArmS,AS,SD), (RSP, MDS,AS, SD), (MDS, OP, AS, MD), (MSC3,MDS,AS,SD), (MSC2,MDS,AS,SD), (ArmS,OP,AS,MD), (MSC3, ArmS, AS, SD), (OP, ArmS, SY, SD), (MSC2, ArmS, AS, SD)]. As a result, Node 8's similarity metric becomes 0.6875. Later in time, Node 8 receives a gossip message from Node 9 with an AML that reflects Node 9's current view of the architecture. This AML is aggregated with Node 8's AML resulting in an AML that reflects the entire software architecture. When the similarity metric for Node 8 is next computed, it shows a value of 1, indicating convergence at Node 8.

To test the scalability of the approach we tested DeSARM on Argo [5], a high performance computing cluster operated by the Office of Research Computing at George Mason University. For that purpose we put together a synthetic application with 30 components, each one of them residing in a different node of the research cluster. Some components export a synchronous interface only, sending and receiving only synchronous messages, some have an asynchronous interface only, sending and receiving only asynchronous messages, while others comprise both synchronous and asynchronous interfaces, sending synchronous messages and receiving asynchronous messages or vice versa. All communication between components take place at a random time intervals to emulate local processing between message exchanges.

Each component communicates with only two other components so that the initial AML at each node is a very small fraction of the complete AML. Therefore, the value of the similarity metric is very small to being with. It starts at zero in some cases, when components on a node send a synchronous message to another component and have to wait for the reply. This way, the DeSARM instance at each node would take longer to learn the communication patterns at each of the other 27 nodes.

The convergence time at the 30 nodes varied significantly due to the randomness in message exchange. The node that took the longest time to converge converged in 260 sec (i.e., 4.3 minutes) as shown in Table 6. This table shows the progression of the similarity metric over time. The table also shows that the rate of convergence, roughly defined as the increase in convergence over time is slower at the beginning and faster at the end. For example, after 58% of the time, the similarity metric has only achieved the value of 0.27. At 85% of the time, the similarity metrics achieved 0.73. While the slowest node to converge took over four minutes, the fastest took 30 seconds.

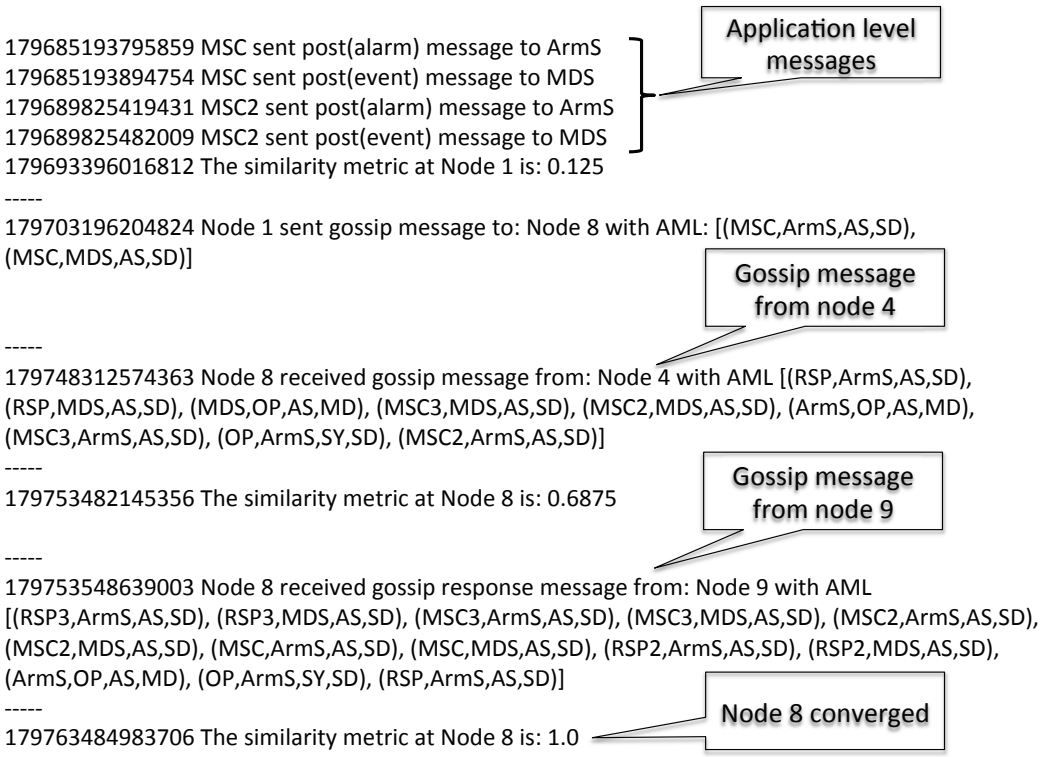

Figure 7: Excerpts of event trace.

Table 6: Slowest convegence rate in the 30-node experiment.

Time (sec)	0-110	120	130	140	150	160-170	180	190-200	210-220	230-250	260
Similarity Metric	0.00	0.10	0.13	0.20	0.27	0.30	0.47	0.57	0.73	0.97	1.00

6 Concluding Remarks

This report presented DeSARM, a completely decentralized and automated approach for software architecture discovery of distributed systems based on gossiping and message tracing. Through message tracing, the technique is able to identify important architectural characteristics such as components and connectors in addition to synchronous and asynchronous communication patterns. Furthermore, through the use of gossiping, DeSARM exhibits the properties of scalability, global consistency among participating nodes, self-organization, and resiliency to failures. These properties were demonstrated through small and large scale experiments, with and without component failures. These experiments assessed the rate of convergence of the DeSARM nodes towards the software architecture being recovered. These experiments showed that DeSARM is resilient and is able to recover the architecture even in the presence of failures, albeit at a lower pace than the one when no failures occur. DeSARM was implemented in Java using a multi-threaded architecture.

We are currently examining how DeSARM can be adapted to recover architectures that change over time as in the case of software dynamic adaptation [19, 18].

Acknowledgements

This work was partially supported by the AFOSR grant FA9550-16-1-0030.

References

[1] Alvisi, Lorenzo, and Keith Marzullo. "Message logging: Pessimistic, optimistic, causal, and optimal." Software Engineering, IEEE Tr., 24.2 (1998): 149-159.

[2] Andritsos, Periklis, and Vassilios Tzerpos. "Information-theoretic software clustering." Software Engineering, IEEE Tr., 31.2 (2005): 150-165.

[3] Anquetil, Nicolas, and Timothy C. Lethbridge. "Recovering software architecture from the names of source files." J. Software Maintenance 11.3 (1999): 201-221.

[4] Antoniol, Giuliano, Massimiliano Di Penta, and Michele Zazzara. "Understanding web applications through dynamic analysis." Program Comprehension, Proc. 12th IEEE Intl. Workshop on. IEEE, 2004.

[5] http://orc.gmu.edu/research-computing/argo-cluster/

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

[7] Corazza, Anna, et al. "Investigating the use of lexical information for software system clustering." Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. IEEE, 2011.

[8] Corazza, Anna, Sergio Di Martino, and Giuseppe Scanniello. "A probabilistic based approach towards software system clustering." Software Maintenance and Reengineering (CSMR), 2010 14th European Conf. IEEE, 2010.

[9] Demers, Alan, et al. "Epidemic algorithms for replicated database maintenance." Proc.Sixth Annual ACM Symp. Principles of distributed computing. ACM, 1987.

[10] De Silva, Lakshitha, and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey." J. Systems and Software 85.1 (2012): 132-151.

[11] Ducasse, Stphane, and Damien Pollet. "Software architecture reconstruction: A process-oriented taxonomy." IEEE Tr. Software Engineering 35.4 (2009): 573-591.

[12] Dulman, Stefan, and Eric Pauwels. "Self-Stabilized Fast Gossiping Algorithms." ACM Tr. Autonomous and Adaptive Systems (TAAS) 10.4 (2015): 29.

[13] Esfahani, Naeem, Eric Yuan, Kyle R, Canavera and Sam Malek. "Inferring Software Component Interaction Dependencies for Adaptation Support." ACM Tr. Autonomous and Adaptive Systems (TAAS) 10.4 (2016): 26.

[14] Ewing,J. and D.A. Menascé, "A Meta-controller method for improving run-time self-architecting in SOA systems," Proc. 5th ACM/SPEC Intl. Conf. Performance Engineering (ICPE 2014), Dublin, Ireland, March 23-26, 2014.

[15] Foggia, Pasquale, Carlo Sansone, and Mario Vento. "A performance comparison of five algorithms for graph isomorphism." Proc. 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition. 2001.

[16] Garcia, Joshua, Igor Ivkovic, and Nenad Medvidovic. "A comparative analysis of software architecture recovery techniques." , 2013 IEEE/ACM 28th Intl. Conf. Automated Software Engineering (ASE).

[17] Garcia, Joshua, et al. "Enhancing architectural recovery using concerns." Proc. 2011 26th IEEE/ACM Intl. Conf. Automated Software Engineering, 2011.

[18] Gomaa, H., Hashimoto, K., 2012. "Dynamic self-adaptation for distributed service-oriented transactions, '' Proc. 7th Intl. Symp. Softw. Eng. for Adaptive and Self-Managing Systems, SEAMS 12. IEEE Press, Piscataway, NJ, USA, pp. 1120.

[19] Gomaa, H., Hashimoto, K., Kim, M., Malek, S., Menascé, D.A., 2010. "Software adaptation patterns for service-oriented architectures," Proc. 2010 ACM Symp. Applied Computing, New York, NY, USA, pp. 462469. doi:10.1145/1774088.1774185.

[20] Harris, David R., Howard B. Reubenstein, and Alexander S. Yeh. "Reverse engineering to the architectural level." Proc.17th Intl. Conf. Software engineering, ACM, 1995.

[21] Holt, Richard C., et al. "GXL: A graph-based standard exchange format for reengineering." Science of Computer Programming 60.2 (2006): 149-170.

[22] Huang, Gang, Hong Mei, and Fu-Qing Yang. "Runtime recovery and manipulation of software architecture of component-based systems." Automated Software Engineering 13.2 (2006): 257-281.

[23] Israr, Tauseef, Murray Woodside, and Greg Franks. "Interaction tree algorithms to extract effective architecture and layered performance models from traces." J. Systems and Software 80.4 (2007): 474-492.

[24] Johnson, David B., and Willy Zwaenepoel. "Recovery in distributed systems using asynchronous message logging and checkpointing." Proc. 7th Annual ACM Symp. Principles of distributed computing. ACM, 1988.

[25] Kephart, Jeffrey O., and David M. Chess. "The vision of autonomic computing." IEEE Computer 36.1 (2003): 41-50.

[26] Kermarrec, Anne-Marie, and Maarten Van Steen. "Gossiping in distributed systems." ACM SIGOPS Operating Systems Review 41.5 (2007): 2-7.

[27] Koschke, Rainer. "Architecture reconstruction." Software Engineering. Springer Berlin Heidelberg, 2009. 140-173.

[28] Koutra, Danai, et al. Algorithms for graph similarity and subgraph matching. Technical Report Carnegie-Mellon-University, 2011.

[29] Kuhn, Adrian, Stphane Ducasse, and Tudor Grba. "Semantic clustering: Identifying topics in source code." Information and Software Technology 49.3 (2007): 230-243.

[30] Langelier, Guillaume, Houari Sahraoui, and Pierre Poulin. "Visualization-based analysis of quality for large-scale software systems." Proc. 20th IEEE/ACM Intl. Conf. Automated software engineering. ACM, 2005.

[31] Lethbridge, Timothy C., Sander Tichelaar, and Erhard Pldereder. "The dagstuhl middle metamodel: A schema for reverse engineering." Electronic Notes in Theoretical Computer Science 94 (2004): 7-18.

[32] Lungu, Mircea, Michele Lanza, and Tudor Grba. "Package patterns for visual architecture recovery." Software Maintenance and Reengineering, 2006. CSMR 2006. Proc. 10th European Conf. IEEE, 2006.

[33] Maqbool, Onaiza, and Haroon A. Babri. "Hierarchical clustering for software architecture recovery." IEEE Tr. Software Engineering, 33.11 (2007): 759-780.

[34] Marcus, Andrian, et al. "An information retrieval approach to concept location in source code." Proc.11th IEEE Working Conf. Reverse Engineering, 2004.

[35] Menascé, D.A., J. Ewing, H. Gomaa, S. Malex, and J.P. Sousa, "A framework for utility-based service-oriented design in SASSY," First Joint WOSP/SIPEW Intl. Conf. Performance Engineering, San Jose, CA, USA, January 28-30, 2010.

[36] Menascé, D.A., and L. Kanchanapalli. "Probabilistic scalable P2P resource location services." ACM SIGMETRICS Performance Evaluation Review 30.2 (2002): 48-58.

[37] Menascé, D.A., et al. "SASSY: A framework for self-architecting service-oriented systems." Software, IEEE 28.6 (2011): 78-85.

[38] Mendonça, Nabor C., and J. Kramer. "Architecture recovery for distributed systems." SWARM Forum at the Eight Working Conf. Reverse Engineering. 2001.

[39] Mirakhorli, Mehdi. "Software architecture reconstruction: Why? What? How?." Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd Intl. Conf. Software Analysis, Evolution and Reengineering, 2015.

[40] Misra, Janardan, et al. "Software clustering: Unifying syntactic and semantic features." , 2012 19th IEEE Working Conf. Reverse Engineering (WCRE), 2012.

[41] Murphy, Gail C., David Notkin, and Kevin Sullivan. "Software reflexion models: Bridging the gap between source and high-level models." ACM SIGSOFT Software Engineering Notes 20.4 (1995): 18-28.

[42] Naseem, Rashid, Onaiza Maqbool, and Siraj Muhammad. "Improved similarity measures for software clustering." , 2011 15th IEEE European

Conf. Software Maintenance and Reengineering (CSMR), 2011.

[43] Pinzger, Martin, and Harald Gall. "Pattern-supported architecture recovery." , 2002. Proc. 10th Intl. IEEE Workshop on Program Comprehension. 2002.

[44] Pinzger, Martin, et al. "Revealer: A lexical pattern matcher for architecture recovery." , 2002. Proc. 9th IEEE Working Conf. Reverse Engineering, 2002.

[45] Pinzger, Martin, Harald Gall, and Michael Fischer. "Towards an integrated view on architecture and its evolution." Electronic Notes in Theoretical Computer Science 127.3 (2005): 183-196.

[46] Qingshan, Li, et al. "Architecture recovery and abstraction from the perspective of processes." Reverse Engineering, 12th Working Conference on. IEEE, 2005.

[47] Riva, Claudio, and Jordi Vidal Rodriguez. "Combining static and dynamic views for architecture reconstruction." Software Maintenance and Reengineering, 2002. Proc. 6th European Conf. IEEE, 2002.

[48] Riva, Claudio, and Yaojin Yang. "Generation of architectural documentation using XML." Reverse Engineering, 2002. Proc. 9th Working Conf.. IEEE, 2002.

[49] Riviere, Etienne, and Spyros Voulgaris. "Gossip-based networking for internet-scale distributed systems." E-Technologies: Transformation in a Connected World. Springer Berlin Heidelberg, 2011. 253-284.

[50] Sartipi, Kamran, and Nima Dezhkam. "An Amalgamated Dynamic and Static Architecture Reconstruction Framework to Control Component Interactions 259." Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on. IEEE, 2007.

[51] Sartipi, Kamran. "Alborz: A Query-based Tool for Software Architecture Recovery." IWPC. 2001.

[52] Sartipi, Kamran. "Software architecture recovery based on pattern matching." Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on. IEEE, 2003.

[53] Schmerl, Bradley, et al. "Discovering architectures from running systems." Software Engineering, IEEE Transactions on 32.7 (2006): 454-466.

[54] Siff, Michael, and Thomas Reps. "Identifying modules via concept analysis." Software Engineering, IEEE Transactions on 25.6 (1999): 749-768.

[55] Sistla, A. Prasad, and Jennifer L. Welch. "Efficient distributed recovery using message logging." Proceedings of the eighth annual ACM Symposium on Principles of distributed computing. ACM, 1989.

[56] Taylor, Richard N., Nenad Medvidovic, and Eric M. Dashofy. Software architecture: foundations, theory, and practice. Wiley Publishing, 2009.

[57] Tzerpos, Vassilios, and Richard C. Holt. "ACDC: An algorithm for comprehension-driven clustering." wcre. IEEE, 2000.

[58] Vasconcelos, Aline, and Cludia Werner. "Software architecture recovery based on dynamic analysis." Brazilian Symp. on Softw. Engineering. 2004.

[59] Weyns, Danny, and Tanvir Ahmad. "Claims and evidence for architecture-based self-adaptation: A systematic literature review." Software Architecture. Springer Berlin Heidelberg, 2013. 249-265.

[60] Wiggerts, Theo A. "Using clustering algorithms in legacy systems remodularization." Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on. IEEE, 1997.

[61] Wu, Xiaomin, et al. "A reverse engineering approach to support software maintenance: Version control knowledge extraction." Reverse Engineering, 2004. Proceedings. 11th Working Conference on. IEEE, 2004.

[62] Yeh, Alexander S., David R. Harris, and Melissa P. Chase. "Manipulating recovered software architecture views." Proceedings of the 19th international conference on Software engineering. ACM, 1997.

[63] Yuan, Eric, and Sam Malek. "Mining Software Component Interactions to Detect Security Threats at the Architectural Level." Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture. 2016.

[64] Zager, Laura A., and George C. Verghese. "Graph similarity scoring and matching." Applied mathematics letters 21.1 (2008): 86-94.