Mining Approach for Software Architectures' Description Discovery

Mariam Chaabane, Ismael Bouassida Rodriguez, Khalil Drira, Mohamed Jmaiel

▶ **To cite this version:**

Mariam Chaabane, Ismael Bouassida Rodriguez, Khalil Drira, Mohamed Jmaiel. Mining Approach for Software Architectures' Description Discovery. IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), Oct 2017, Hammamet, Tunisia. 12p. hal-01731365

HAL Id: hal-01731365

https://laas.hal.science/hal-01731365v1

Submitted on 14 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Mining Approach for Software Architectures' Description Discovery

Mariam Chaabane*
ReDCAD Laboratory
University of Sfax
Sfax, Tunisia

Ismael BOUASSIDA RODRIGUEZ
ReDCAD Laboratory
Digital Research Center of Sfax
Sfax, Tunisia

Khalil DRIRA
University of Toulouse
CNRS, LAAS
Toulouse, France

Mohamed JMAIEL
ReDCAD Laboratory
Digital Research Center of Sfax
Sfax, Tunisia

Abstract

System of Systems (SoS) is a new class of complex software systems resulting from the integration of several independent systems working together. Within a SoS, many participant systems may be integrated and deleted operationally over the time. Each system has an Architecture Model *modeled at design time. Thus, the SoS' software architecture description is represented by an aggregated* Architecture Model*. This aggregated* Architecture Model *represents participant systems but not necessarily their interactions and communications over the time. In literature, several research studies addressed issues related to SoS. However, we noticed a lack of studies that address the problem of how to describe the whole SoS' software architecture for each change of a participant system over the time. Moreover, studies dealing with checking conformity between the whole SoS' software architecture description and the aggregated* Architecture Model*, are still lacking. This paper presents an approach for the discovery of SoS' software architecture description from execution traces. For this purpose, the proposed approach records execution traces of all participant systems belonging to the SoS, their interactions and communications in a data base. Then, our approach relies on mining techniques to extract software architecture from the data base and describes it via a model called* Architecture Model*. In addition, this paper offers a solution for checking conformity between the aggregated* Architecture Model *and the* Discovered Model*. The diagnosis results may suggest new rules/constraints to enhance the aggregated* Architecture Model

I. Introduction

The massive development of communication technologies and the integration of several independent systems working together have led to the appearance of large and complex software systems referred as Systems of Systems(SoS). SoS have arisen as a result of the integration of various independent systems, developed with different technologies and for diverse purposes, too[11].

*Corresponding author: mariam.chaabane@redcad.org

Operationally integration of these independent systems provide more complex functions which could not be provided by any only existent system. Since each system is designed and developed separately, the software *Architecture Model* of each system represents only its functionalities. So, in a SoS, the architecture description is an aggregation of *Architecture Models*. Because of the operational integration, the aggregated *Architecture Model* may be not able to represent the provided functionalities and interactions within the SoS. In fact, there

is a lack of consensus on how better dealing with SoS software architecture descriptions, as well as with aspects of the SoS architectural descriptions that require further investigation [6].

Our approach aim to discover the whole SoS' *Architecture Model* from execution traces. Hence, the main contribution of this paper is making the execution traces visible by monitoring the communications of all the actors of SoS and recording the intercepted communications on a data base. Then, the proposed approach entitled *Architecture Mining* allows to discover and to mine the software architecture of the whole SoS from the execution traces recorded on the data base and to represent it as a model called *Discovered Model*. The second step of our approach allows verifying conformity between the *Discovered Model* and the aggregated *Architecture Model* composed from the separated *Architecture Models* of participant systems. The third step intends to enhance the aggregated *Architecture Model* based on alignment with the *Discovered Model*.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 presents a motivating scenario. Section 4 introduces our approach entitled *Architecture Mining*. In Section 5 we explain how we applied our approach on the motivating scenario.

II. Related Work

In literature, there are several studies dealing with discovering the actual architecture from a source code for checking conformity with the designed model. Nevertheless, it is observed that there is a lack of research studies that present solutions to discover architecture from execution traces.

The approach of Caracciolo et al. [3] proposes a Domain Specific Language (DSL) to describe architecture at design time. To unify the functionality provided by existing tools for checking conformity between the *de facto* architecture and its theoretical counterpart, the author proposes a tool coordination framework that verifies rules written with DSL.

Meffort et al. [10] used UML as a description language. They relied on data mining techniques to extract the actual architecture from a source code. In addition, they proposed a methodology that uses the extracted architecture patterns and provided an algorithm of checking conformity that detects both absences and divergences. As result, they identified many architectural violations proving that only 40% were confirmed by a senior software developer.

Weinreich et al. [14] proposed an architecture meta-model, called the LISA model, for describing heterogeneous component-based software architectures. The LISA model supports various concepts for dealing with the typical characteristics of SOA. The author extended this work [13] to Checking conformity between the architecture extracted from the source code and the LISA model. They presented an approach for automatic reference architecture conformance checking of SOA-based software systems.

The approach of Alshara et al. [1] deals with recovering software architecture from object-oriented source code. This approach propose transforming object-oriented code to component-based one guided by the recovered architecture of the corresponding object-oriented software. This approach allows to reveal component-based architecture to materialize the recovered architecture.

For checking conformity of a dynamic software architectures, Cavalcante et al. [4] propose a Statistical Model Checking (SMC). The authors introduce a novel notation to formally express architectural properties as well as an SMC-based toolchain for verifying dynamic software architectures described in pi-ADL. For this purpose, they used a flood monitoring system.

We note that all previous approaches discovered the software architecture from the source code. A key limitation of these research is that they does not address the problem of SoS' software architecture discovery. As many systems may be integrated or deleted over the time in SoS, the previous techniques don't allows to

discover the SoS architecture for each change of a system belonging to the SoS.

The approach of Loukil et al. [9] investigated the extraction of a software architecture from execution traces at runtime to obtain a 'Model at runtime'. At design time, the authors use the Architecture Analysis and Design Language (AADL) for describing the architecture of dynamically adaptive component-based systems. Nevertheless, the proposed checking conformity model can't detect absence cases and doesn't allow recording execution traces in the data base to check conformity with the AADL model.

We observe that most of these studies use declarative language for architecture description. In fact, according to various studies, ADLs fall short in fulfilling requirements such as extensibility, usability and multifaceted modeling. According to Caracciolo et al. [3], ADLs provide poor support for extensibility and neglect aspects that are of most concern to stakeholders. It needs to be simple and intuitive enough to communicate the right message to the stakeholders involved in the architecting phase. Otherwise, many ADLs do not support multiple viewpoints.

Other than ADL, Bouassida et al. [2] provide generic and scalable solutions of architecture's description for automated self-reconfiguration systems. The author elaborates a graph-based modelling approach where vertices correspond to deployment nodes and software services. On the other hand, the author uses rule-oriented techniques, such as graph grammar productions and graph transformation to specify rules for changing deployment architecture while being in conformance with the architectural style.

Through our study, we noted that most of previous studies extract the software architecture from source code and do not take into account SoS characteristics. In fact, in SoS, the aggregation of systems within an SoS may lead to new functionalities. Moreover, many systems may be integrated or deleted operationally over the time. So the aggregated *Architecture Model* will be incomplete for each change. In this study, a new technique to recover the architecture of an SoS is suggested. To discover the *Architecture Model* of participating systems on SoS, our approach proposes the mining of the software architecture of the whole SoS from the execution traces recorded in the data base.

Although, the mining in data field, called 'Data Mining', is a set of methods and techniques to extract useful information through rapid and efficient discovery of unknown or hidden information inside large databases. [8]. In the process field, 'Process Mining' is to discover, monitor and improve real processes by extracting knowledge from the event log. It describes a family of analysis techniques exploiting the information recorded in the event log [7][12].

Taking inspiration from data mining and process mining techniques that consist in the extraction of knowledge from large databases or from events logs, we defined our approach that consists on mining actual SoS' software architecture, entitled *Architecture Mining*. To design the *Architecture Model* of each participant system, we chose graphs as architecture description language. According to this choice, we decide to use research of homomorphism in graphs to detect both absences and divergences between the aggregated *Architecture Model* of the SoS and the *Discovered Model* mined from the data base.

III. Architecture Mining Approach

In this section, we present our approach of *Architecture Mining* and we detail its three steps, namely, Discovery, Conformance and Enhancement.

Figure 1 illustrates the basic elements of our approach. At design time, an *Architecture Model* that is supposed to be respected at runtime is designed. In the case of SoS, participant systems are integrated or deleted from operational viewpoint over time. Thus, the SoS' aggregated *Architecture Model* represents only participant systems but don't illustrates all provided functionalities, interactions and communications

within the SoS.

Our approach consists in deploying a monitor next to each component belonging to participant systems in the SoS. These monitors intercept the messages exchanged between all the components and record the recovered data in the data base to extract the whole model of the SoS. These monitors are software entities used to intercept messages and enrich them with personalized data according to the need of the developer such as the time of sending a request, the component name and the occurrence of sending a request.

At runtime, the monitored execution traces of the SoS are sequentially recorded in the data base. The first step of *Architecture Mining* is discovery. It aims to discover and mine the software architecture of all parts of the SoS from the data base. The second step is Conformance and the third is Enhancement.

i. Discovery

The first step of *Architecture Mining* is discovery. In fact, monitors deployed in each component of the SoS, intercept exchanged messages between different components of running systems and record them in a data base. So, monitoring allows us to obtain execution traces.

The starting point of the discovery step is collecting events and correlating events from execution traces to isolate end-to-end instances of the same scenario. This generates an execution trace corresponding to a scenario instance. An execution trace is a sequentially recorded collection of events, where each event refers to a message and is related to a particular component.

In the discovery step, mining techniques collect data from the data base to represent the *Discovered Model* in terms of a Petri net, Business Process Modeling Notation, Graphs, etc. This step allows to extract monitored events from the data base, discover the actual architecture of SoS and illustrate it by a model called *Discovered Model*. This step makes possible the discovery of all parts of the SoS, their interactions, communications and to obtain a *Discovered Model* from the execution traces. The *Discovered Model* may be used for discussing problems among stakeholders of different participant systems to have a shared view of the real model.

ii. Conformance

The second step of *Architecture Mining* is conformance. This step aims to check if reality, expressed by the *Discovered Model*, conforms with the aggregated *Architecture Model*. Conformance is used likewise to diagnose the differences between the mined behavior represented by the *Discovered Model* and the modeled behavior represented by the aggregated *Architecture Model*. This diagnosis allows us to locate the missing elements and relations between the components belonging to the SoS in the aggregated *Architecture Model*.

The step of conformance checking uses the agregated*Architecture Model* and the *Discovered Model* as input. For conformance checking, the modeled behavior and the observed behavior are compared. Conformance checking techniques relate events in the data base to messages and components in the model.

Conformance checking can be used to identify deviating cases and SoS' parts where most deviations occur and to locate the missing elements and relations between the components within the SoS in the aggregated *Architecture Model*. In addition, this step allows us to continuously evaluate the quality of new *Discovered Model* using conformance checking. Finally, conformance checking is used as a starting point for the third step which is enhancement.

iii. Enhancement

The third step of *Architecture Mining* is enhancement. This step aims to enrich and extend the aggregated *Architecture Model* with the mined behaviors from execution traces represented as a *Discovered Model*. So, the enhancement step can enrich and extend the aggregated *Architecture Model* of the SoS. This step is based on diagnosis' results of the differences between the

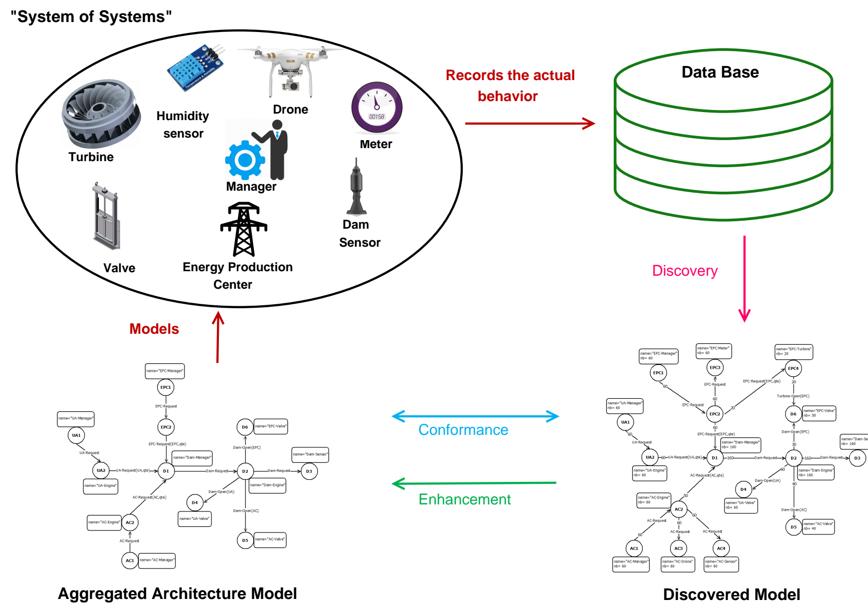

Figure 1: *Architecture Mining*

aggregated *Architecture Model* and the *Discovered Model*. Moreover, monitoring results may contain information about timestamps and the occurrence of each request.

Thus, the aggregated *Architecture Model* of the SoS can be corrected using the diagnostics provided by the alignment with the *Discovered Model*. After alignment, it is possible to replay the discovery step.

IV. Motivating Scenario

This section presents a scenario that will be reused throughout the paper to specify and illustrate our contribution. This scenario is an example of SoS dealing with smart grids and taking place in the agricultural domain. This SoS is the result of the integration of four independent systems, which were developed separately and at different periods.

The first system is the dam. Initially, the dam provides water for an Agricultural Center (AC) which is responsible for an irrigation system. The Agricultural Center is the second participant system in our SoS. Then, channels of potable water are installed from the tank of the dam to the city. The potable water is managed by the Utilities Agency (UA). So a third system is added. Later, a fourth system interferes to take another part of water from the dam's tank. This system is an Energy Production Center (EPC) that needs to produce electricity from a stream of moving water coming from the dam's tank and going through a hydraulic turbine. Thus, we obtain four systems participating in our SoS, each one of them having a different objective.

The main system in our case study is the dam. It is modeled to manage the need of water upon request. The dam has three levels that describe the state of the water in the tank: overfull, medium and drought. Each participant system has a priority degree. The UA has the highest priority, the AC has the second priority and the EPC has the third and last priority. The software architecture relative to the dam considers a rule which takes the water level and the priority of the system as input and decides to accept the request, to reduce the required amount of water or to decline the request. A definite amount of water is taken daily, upon request, for the Utilities Agency even in case of drought. An amount of water is given, upon request, to the Agricultural Center to be shared

among different irrigated fields only in cases of an overfull or a medium level of water in the tank. An amount of water is provided to the Energy Production Center, upon request, only in case of overfull. The managers of Utilities Agency, Agricultural Center and Energy Production Center send request to the dam manager to provide the required amount of water and the system priority. To respond to each request, the dam manager consults the water level in the tank and decides to open the associated valve or to decline the request. If more than a request is sent to the dam, its manager must take a decision according to the priority of each system and the level of water in the tank. It can accept all requests, decline some requests, accept partially the request by reducing the amount of required water, etc. Finally, the sum of provided amounts of water must not exceed a defined volume daily.

In this case study, many system are integrated over the time. Thus, we know only their components, objectives and software architecture separately but not their behaviours, interactions and rules within the SoS.

The Utilities Agency aims to obtain the required amount of water for potation. The Agricultural Center intends to obtain more water for irrigation. The Energy Production Center aims to increase energy production. The dam manager's role is to manage water in the tank. Six components belong to the dam system: a manager, an engine, a valve assigned to the Utilities Agency, a valve assigned to the Agricultural Center, a valve assigned to the Energy Production Center and a sensor to detect water level in the tank. The UA has two component: a manager and an engine. The AC has four components: a manager, an engine, humidity sensors and drones that include a GPS chip and a thermal camera. Thus, they can fly over the field to detect the parts that need to be watered. The EPC has four components: a manger, an engine, a hydraulic turbine producing electricity from a stream of moving water coming from the dam when the associated valve is opened, and a meter that knows the need of electricity.

In the Utilities Agency, the manager triggers the UA engine daily. This one sends a request to the dam manager to open the UA valve. In the Agricultural Center, the manager triggers the AC engine daily. This one asks the AC sensors and the AC drones if the humidity indicator is down. If that is the case, the AC engine sends a request to the dam manager to open the AC valve. In the Energy Production Center, the manager triggers the EPC engine daily. This one asks the EPC meter if it's a lack of electricity. In case of requirement of energy, the EPC engine sends a request to the dam manager to open the EPC valve. The dam's manager, who manages diverse requests of water, can accept the request and open the associated valve, decline the request or reduce the required amount of water, based on water levels provided by the Dam sensor and the system priority.

After our SoS has been operating for a certain period, the dam's manager notices that he can't always satisfy the AC's demands because the daily allowed volume of water is reached although the water level is medium. Since the aggregated *Architecture Model* of the whole SoS don't represent interactions between participant systems within the SoS, we can not discover the problem leading to this observation. Moreover, more systems may be added to this SoS in the future, other may be deleted. Thus, we need to know the software architecture of the SoS for each change over the time.

As mentioned earlier, our approach entitled *Architecture Mining* aims to discover the software *Architecture Model* of the whole SoS to represent interactions and behaviours within the SoS and locate each emergent problem. The proposed approach intends to record the execution traces of all participant systems of the SoS at runtime in the data base. Then, our approach aims at using these execution traces to mine the deployment architecture of the whole SoS, and to locate the problems leading to the wrong management of water and each other problems.

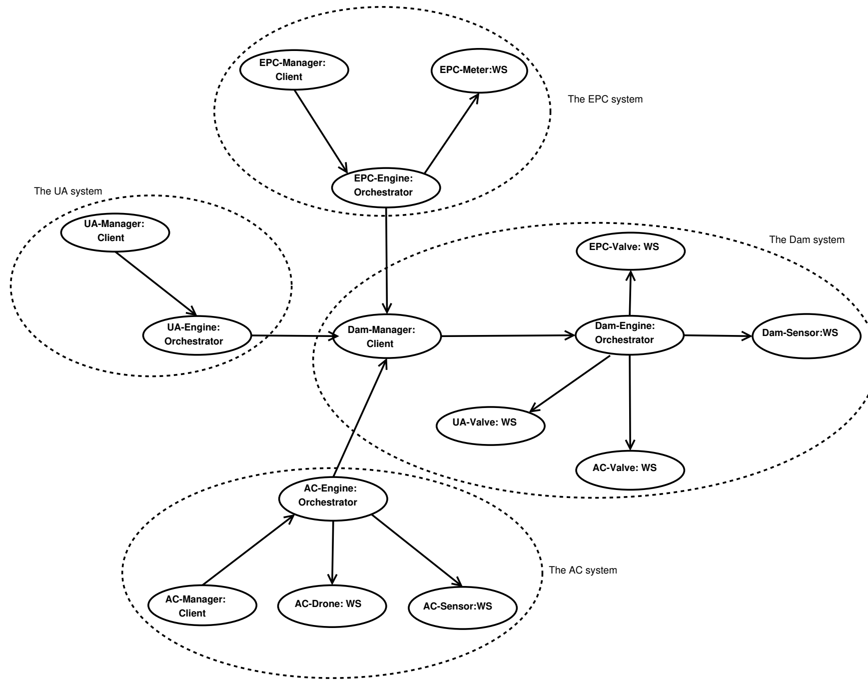

Figure 2: *Case Study Implimenting*

V. Experiments

To illustrate the motivating case study mentioned above, a simulation was performed based on SOA technologies. To implement this simulation, we have used a composition of Web Services by orchestration [5]. We used eclipse development environment that is based on Java EE. We profit of the benefits of the combined use of Tomcat Server and implementation of Axis2 Web Services. Thus, the Web Services orchestrators execute the described process by BPEL language. This allows to invoke atomic Web Services in order to obtain composed Web Service. The composed Web Service is equally presented in WSDL but deployed in Apache-Ode, which is deployment engine for orchestration process.

The type of service-oriented application that we have developed is composed of three types of entities: the atomic Web Services (WS), Web Services orchestrators (Orchestrator) and Web Services clients (Client). As indicated in Figure 2, the UA Valve, the AC Valve, the EPC Valve, the Dam Sensor belonging to the Dam system, the Meter belonging to the EPC system, the Sensors and the Drones belonging to the AC system are represented by WS. The Dam Engine, the UA Engine, the AC Engine and the EPC Engine are represented by Orchestrators. The Dam Manager, the UA Manager, the AC Manager and the EPC Manager are represented by Clients.

Then, we deployed monitors next to each entity belonging to the SoS as indicated in Figure 2. The monitors intercept the SOAP messages exchanged between the entities and save the recovered data in the database to mine the whole architecture of the proposed SoS.

A monitor is a software entity used to intercept SOAP messages [5]. It enriches SOAP messages with personalized data according to the need of the user such as the identifier and the name of the message, the identifier and the name of the source and target Web Service, etc. We implemented a Web Service side monitor, an Orchestrator side monitor and a Client side monitor. The Client side monitor intercepts messages exchanged between the Client and the Orchestrator. The Web Service side monitor

intercepts the messages exchanged between the Orchestrator and the Web Service. The Orchestrator side monitor intercepts all the messages entering and leaving the Orchestrator. All the monitors communicate with the database to save the data resulting from the interception of the messages.

To simulate the software architecture of our SoS, we have executed the test scenario mentioned in section IV, 180 times.

i. Discovery

To recover the software architecture of our SoS, we mined data resulting from the interception of the messages stored in the database during the execution of the test scenario 180 times. After mining the data recorded in the database, we performed some calculations to obtain the necessary data to generate the *Discovered Model*, namely the invocation number of each entity (WS, Orchestrator, Client) and the execution number of each operation. These calculations allow us to know the occurrence of each communication.

In the discovery step, mining algorithm collects data from the data base to represent the *Discovered Model* in terms of a Petri net, Business Process Modeling Notation, Graphs, etc. In this paper, we chose to describe the *Discovered Model* as a graph which represent a powerful and versatile tool for modelling real systems in diverse domains such as distributed systems[2]. In the proposed graph, a node corresponds to an entity that interacts with another; a link illustrates the flow of the messages transmitted between two nodes. Each node is represented by: the entity ID displayed on the node, the entity name and the invocation number of the entity. Each link is illustrated by: the operation name(parameter) and the execution number of the operation.

To obtain such a graph, we recorded mined and collected data in an xml file. Each entity is illustrated by a $< node >$ tag. The entity's ID is displayed on the node, the entity name and the invocation number of the entity are represented as attributes for the $< node >$ tag. Each link is illustrated by an $< edge >$ tag. The operation name, the execution number of the operation, the source node and the target node are represented as attributes for the $< edge >$ tag.

Finally, we need to illustrate graphically the *Discovered Model* and convert the *Discovered Model* from xml form to a graphic form. So, we generated a graph from the xml file using the open source Java module entitled 'Blueprints'. This module uses an xml file as input and provides a labelled graph as output. We obtained the graph illustrated in Figure 3 that illustrates the execution of our test scenario 180 times.

We noticed first of all that the Dam Manager can't satisfy all the requests of the AC Engine and the EPC Engine. In Figure 3, the AC Engine sends a request to the Dam Manager *'AC-Request(AC, qte)'* 50 times, but the Dam Engine opens the AC Valve *'Dam-Open(AC)'* only 40 times. Likewise, the EPC Engine sends a request to the Dam Manager *'EPC-Request(EPC, qte)'* 50 times, but the Dam Engine opens the EPC Valve *'Dam-Open(EPC)'* only 30 times. Although the operation *'Dam-Open(EPC)'* is executed only 30 times by the Dam Engine to open the EPC Valve as shown in the message on the link, the EPC Valve is invoked 50 times, as shown in the message on the node *'D6, EPC-Valve, 50'*. So the EPC Valve is opened 20 times by another entity.

The first step of *Architecture Mining* called Discovery makes possible the discovery of the whole SoS's architecture through the mining of the execution traces recorded in the database. Based on the conformance checking, the designer can diagnose the differences between the *Discovered Model* and the aggregated *Architecture Model* and enhance the software architecture of the SoS by adding more rules.

ii. Conformance

Taking inspiration from the process mining field [12], we applied the conformance checking techniques to diagnose the differences between the *Discovered Model* and the aggregated *Architecture Model* of our SoS.

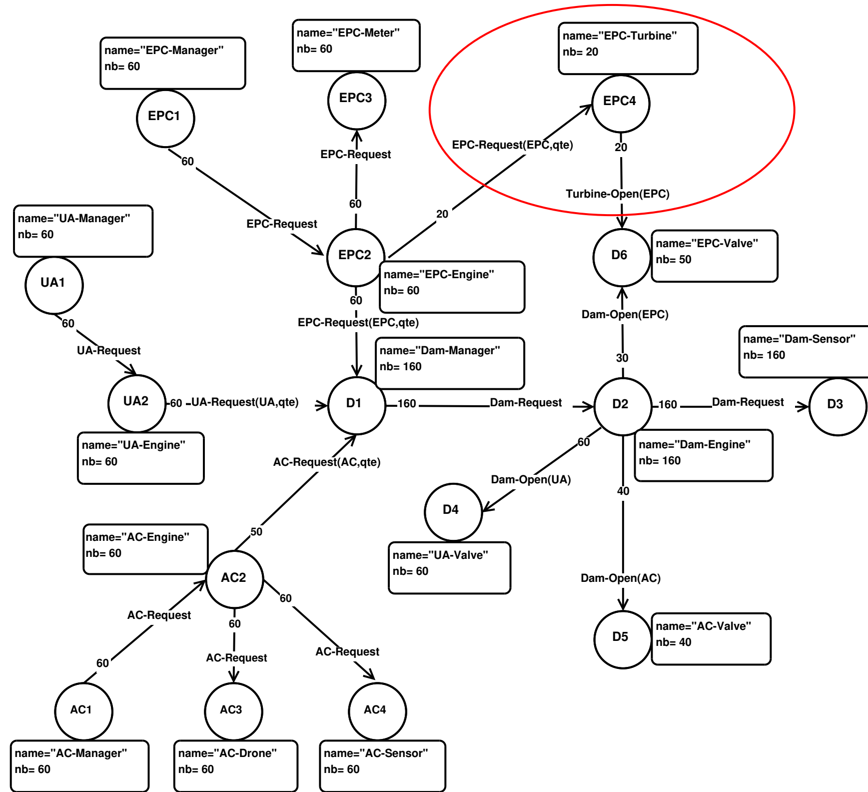

Figure 3: *Discovered Model*

For this purpose, we applied the footprint algorithm [12] which is a matrix showing causal dependencies between entities and interactions. For example, the footprint of an execution trace may show that the node x receives sometimes a request from the node y but never the other way around. If the footprint of the aggregated *Architecture Model* shows that x never receives a request from y, then the footprints of the execution trace and the aggregated *Architecture Model* disagree on the ordering relation of x and y.

When observing the *Discovered Model*, we notice that there is a link between the EPC engine's node and another node called *'EPC-Turbine'*, and the link between this node and the EPC valve's node. These links illustrate a request from the EPC Engine to the EPC-Turbine *'EPC-Request(EPC, qte)'* to open the EPC valve without the permission of the Dam Engine. According to the *Discovered Model* illustrated in Figure 3, the EPC-Turbine executed an operation called *'Turbine-Open(EPC)'* to open the EPC Valve 20 times. However, the request *'Turbine-Open(EPC)'* does not exist in the aggregated *Architecture Model* illustrated in Figure 4.

The conformance algorithm allows us to locate the problem and to know why the volume of water allowed daily is reached before the Dam Manager can respond to the requests of the AC Engine and the EPC Engine. In fact, the monitor deployed in the EPC engine intercepts the communication between the EPC Engine and the EPC-Turbine. Likewise, the monitor deployed next to the EPC Valve intercepts the communication between the EPC-Turbine and the EPC Valve. These monitors discover the EPC-Turbine node and their communications with the other entities in our SoS.

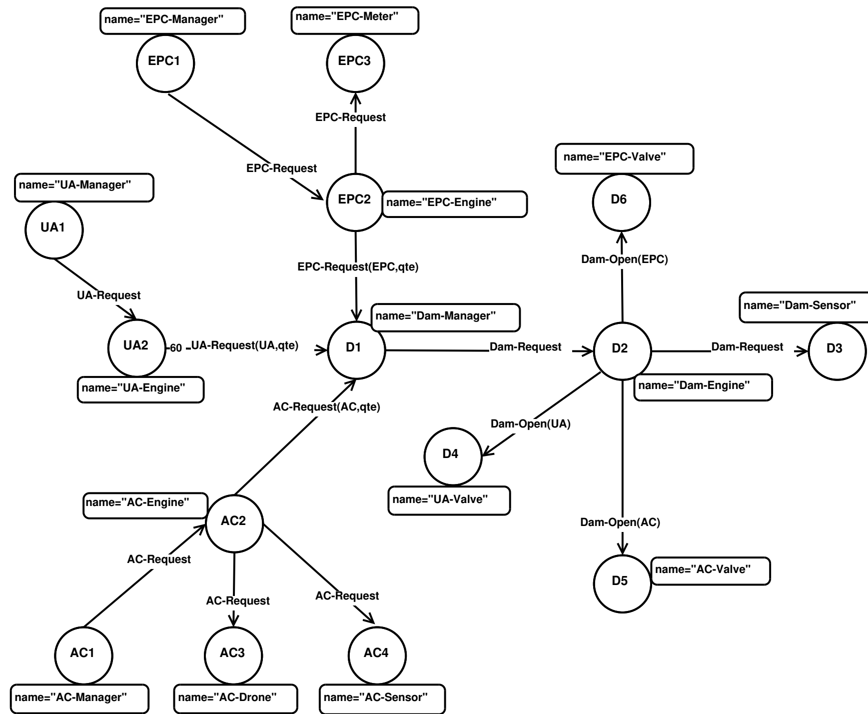

Figure 4: *Aggregated Architecture Model*

iii. Enhancement

The third step of *Architecture Mining* allow us to extend or improve the aggregated *Architecture Model*. This step is based on the diagnostics provided by the alignment of the aggregated *Architecture Model* and the *Discovered Model*. Moreover, monitored data may contain information about timestamps and the occurrence of each communication.

The alignment of the aggregated *Architecture Model* and the *Discovered Model* allows us to enrich the aggregated *Architecture Model*. Thus, we propose a constraint that prohibits the *'EPC-Turbine'* from sending a request *'Open(EPC)'* to the EPC Valve. This rule restrict sending a request *'Open(EPC)'* to any valve for the Dam Engine. Thus, in the new *Architecture Model*, the *'EPC-Turbine'* must never receives a request from the EPC engine. Likewise, the EPC Valve must never receives a request *'Turbine-Open(EPC)'* from the node *'EPC-Turbine'*.

VI. Conclusion

In this paper, we have introduced a new approach entitled *Architecture Mining*. This approach addresses the problem of describing the evolution of SoS' software architecture. For this purpose, we have monitored performed events and recorded execution traces of the SoS in a data base. Particular attention is paid for mining the SoS' software architecture from the data base to illustrate it via a model called the *Discovered Model*. The originality of our solution lies in the fact that it allow the discovery of the actual architecture of SoS for each time when a participant system is integrated or deleted. Moreover, to our knowledge, this is the first study that allows the discovery of interactions between systems belonging to an SoS and not illustrated in the aggregated *Architecture Model*.

This approach has been developed for an SoS dealing with smart grids and taking place in the agricultural domain. This SoS is the result of the integration of four independent systems,

even tough they were developed separately: The dam, which is the main system, the Utilities Agency, the Energy Production Center and the Agricultural Center. We have developed and deployed a monitor next to each entity of the SoS to record the execution traces in the data base.

We applied the first step of *Architecture Mining* called Discovery. Discovery consists in mining the whole SoS' software architecture from the data base and describing it via a *Discovered Model* as a graph. Then, we applied the second and the third steps of *Architecture Mining*, namely conformance checking and enhancement. Conformance checking approves us to diagnose the differences between the *Discovered Model* and the aggregated *Architecture Model* of our SoS, locate and explain the encountered problems and deviations. Enhancement allows us to enrich and extend the aggregated *Architecture Model* with new constraints based on the monitored communications recorded in the data base and represented by the *Discovered Model*.

In future research we intend to apply our approach on a larger SoS to address the scalability issue. Moreover, experiments will be needed to verify if our new mining technique offers the same results with a larger SoS.

References

[1] Z. Al-Shara, A.-D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, and A. Shatnawi. Materializing Architecture Recovered from OO Source Code in Component-Based Languages. In *European Conference on Software Architecture*, 2016.

[2] I. Bouassida Rodriguez, K. Guennoun, K. Drira, C. Chassot, and M. Jmaiel. Implementing a rule-driven approach for architectural self configuration in collaborative activities using a graph rewriting formalism. In *CSTST*, pages 484–491, 2008.

[3] A. Caracciolo, M. F. Lungu, and O. Nierstrasz. A unified approach to architecture conformance checking. In *WICSA*, pages 41–50, 2015.

[4] E. Cavalcante, J. Quilbeuf, L.-M. Traonouez, F. Oquendo, T. Batista, and A. Legay. *Statistical Model Checking of Dynamic Software Architectures*, pages 185–200. 2016.

[5] M. Chaabane, F. Krichen, I. Bouassida Rodriguez, and M. Jmaiel. Monitoring of service-oriented applications for the reconstruction of interactions models. In *ICCSA*, pages 172–186, 2015.

[6] M. Guessi, V. V. G. Neto, T. Bianchi, K. R. Felizardo, F. Oquendo, and E. Y. Nakagawa. A systematic literature review on the description of software architectures for systems of systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1433–1440, 2015.

[7] G. Lakshmanan and R. Khalaf. Leveraging process-mining techniques. *IT Professional*, 15(5):22–30, 2013.

[8] B. N. Lakshmi and G. H. Raghunandhan. A conceptual overview of data mining. In *Innovations in Emerging Technology, National Conference on*, pages 27–32, 2011.

[9] S. Loukil, S. Kallel, and M. Jmaiel. Runtime adaptation of component based systems. In *Proceedings of the first International Conference on Networked Systems*, volume 7853, page 284âĂŞ288, 2013.

[10] C. A. Maffort, M. T. Valente, R. Bigonha, A. Hora, N. Anquetil, and J. Menezes. Mining architectural patterns using association rules. In *SEKE*, pages 375–380, 2013.

[11] E. Y. Nakagawa, M. Gonçalves, M. Guessi, L. B. R. Oliveira, and F. Oquendo. The state of the art and future perspectives in systems of systems software architectures. In *Proceedings of the 1st Int. Workshop on SESOS*, pages 13–20, 2013.

[12] W. Van der Aalst. Process mining: Overview and opportunities. *ACM Trans. Manage. Inf. Syst.*, 3(2):7:1–7:17, 2012.

[13] B. Weinreich. Automatic reference architecture conformance checking for soa-based software systems. In *WICSA*, pages 95–104, 2014.

[14] R. Weinreich, C. Miesbauer, G. Buchgeher, and T. Kriechbaum. Extracting and facilitating architecture in service-oriented software systems. In *WICSA-ECSA*, pages 81–90, 2012.