Software Architectural Model Discovery from Execution Data

Cong Liu[1], Boudewijn van Dongen[1], Nour Assy[1] and Wil M.P. van der Aalst[2,1]
[1]*Department of Mathematics and Computer Science, Eindhoven University of Technology, 600MB Eindhoven, The Netherlands*
[2]*Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany*

1 RESEARCH PROBLEM

Software systems form an integral part of the most complex artifacts built by humans, and we have become totally dependent on these complex software artifacts (van der Aalst, 2015). Communication, healthcare, education and government all rely on software systems that take over more and more duties. Modern enterprises continue to invest in the creation, maintenance and change of complex software systems. However, numerous software projects still experience significant problems. Moreover, the complexity of modern software containing millions of lines of codes and thousands of dependencies among components is extremely high (Rubin et al., 2014), (Liu et al., 2016), (Liu et al., 2018c), (Liu et al., 2018d). This complexity makes it difficult to understand, maintain, evolve, improve, and etc.

During the execution of software systems, many crashes and exceptions may occur, and it is a real challenge to understand how a software system is behaving. By exploiting the data recorded during the execution of software systems, one can discover behavioral models to describe the actual execution of software. The discovered behavioral models provide extensive insights into the real usage of software, enable new forms of model-based testing and improvements. Replaying execution data on such models helps to localize performance problems and architectural challenges.

To help understanding the runtime behavior of a software system, we aim to discover an architectural model from the execution data. An architectural model typically structures a software system in terms of components, interfaces and interactions. Generally speaking, our research aims to answer the following questions:

- How does the software system behave at runtime?
  - How many components are involved during the execution of the software system and how they really behave?
  - How many interfaces does a component contain and do they adhere to a typical (pre-defined) behavioral contract?
  - How do components interact with each other during execution?
  - What does the architectural model discovered from the execution data look like?
- How is the quality of the architectural model that we discovered from the execution data? Does it conform to the reality (i.e., execution data)?

2 OUTLINE OF OBJECTIVES

To answer the research questions, our research aims to target the following challenges:

- Challenge 1: propose automated approaches to discover architectural model from software execution data.
  - Challenge 1.1: propose a standardized format for software execution data exchange.
  - Challenge 1.2: propose effective approaches to support the component identification and behavioral model discovery.
  - Challenge 1.3: propose effective approaches to support the interface identification and contract model discovery.
  - Challenge 1.4: propose effective approaches to support the discovery of architectural models.
- Challenge 2: propose effective approaches to support conformance checking based on the discovered architectural model and execution data.
- Challenge 3: evaluate and validate the applicability and effectiveness of the previous approaches using real-life software cases.

Liu, C., van Dongen, B., Assy, N. and van der Aalst, W.
Software Architectural Model Discovery from Execution Data.
In *Doctoral Consortium (ENASE 2018)*, pages 3-10

3 STATE OF THE ART

In this section, we briefly review the state-of-the-art from the following three perspectives: (1) software dynamic analysis; (2) software process mining; and (3) software architecture reconstruction.

3.1 Software Dynamic Analysis

Software dynamic analysis is used to understand the behavior of software by exploiting its execution data. Several techniques and tools have been presented to extract information from running software. Most existing approaches, such as (Lo et al., 2009) and (Walkinshaw and Bogdanov, 2008), generate automaton-based models using different variants of the *K-Tail* algorithm which was first defined by *Biermann* and *Feldman* (Biermann and Feldman, 1972). However, these techniques cannot discover concurrency explicitly, resulting in a so-called state explosion for complex models. Although automaton-based models are popular in software analysis, there are several other techniques to learn other types of models. For example, some techniques visualize software execution traces as sequence diagrams (McGavin et al., 2006) and some of them are extended with loops (Briand et al., 2006). Similar to automaton based models, the (classic) sequence diagram-based models also lack concurrency description. Moreover, each sequence diagram or automaton-based model only describes the behavior of a single execution trace. Given software execution data referring to thousands of traces, these existing approaches will obtain an excessive number of behavioral models rather than a compact model for the whole data. In addition, considering the hierarchical nature of software, the discovered flat sequence diagrams or flat automation-based models cannot accurately capture the real behavior in a meaningful way.

3.2 Software Process Mining

With the development of process mining (van der Aalst, 2016) on the one hand, and the growing availability of software execution data on the other hand, a new form of software analytics comes into reach, i.e., applying process mining techniques to analyze software execution data. This inter-disciplinary research area is called *Software Process Mining (SPM)* (Liu et al., 2016), (Rubin et al., 2007), and aims to analyze software execution data from a process-oriented perspective. One of the first papers addressing *SPM* is (van der Aalst et al., 2015). For the mining of software systems, the recorded events explicitly refer to parts of the system (components, services, etc.). References to system parts facilitate the generation of localized event logs. A generic process discovery approach is proposed based on such localized event logs. Experimental results show that location information indeed helps to improve the quality of the discovered models.

Leemans and van der Aalst (Leemans and van der Aalst, 2015) discover and analyze the operational processes of software systems using process mining techniques. They propose to discover flat behavioral models using Inductive Miner (Leemans et al., 2013). By taking full consideration of component-based architecture and hierarchical structure of a software system, *Liu et al.* (Liu et al., 2016) propose to discover a hierarchical behavioral model for each component. The discovered component model describes the software behavior from the perspective of individual component. However, this work neglects the functions (interfaces) that each component provides to other components as well as the interaction among components.

3.3 Software Architecture Reconstruction

Software architecture reconstruction aims to abstract, identify, and present high-level views from low-level data to help understanding software. The recovered architectural views play a pivotal role in software understanding, reuse, evolution, maintenance, etc. (Garlan, 2000), (Stéphane and Damien, 2009). *Crnkovic et al.* (Ivica et al., 2011) discuss fundamental principles of software architectural models and compare a large number of existing architectural models, such as *Enterprise JavaBeans*, *Microsoft Component Object Model* and *CORBA Component Model*.

For software systems that are implemented by object-oriented technology, a component is composed of a set of classes and an interface is composed of a set of methods. Various clustering-based techniques are proposed to identify components based on different criteria such as coupling, cohesion and modularity. According to the required input, these approaches can be classified as development documents based approaches (e.g., (Lee et al., 2001), (Kim and Chang, 2004), (Chang et al., 2005), (Hasheminejad and Jalili, 2015)), source code based approaches (e.g., (Washizaki and Fukazawa, 2005), (Kebir et al., 2012a), (Kebir et al., 2012b), (Cui and Chae, 2011), (Luo et al., 2004), (Chiricota et al., 2003), (Mancoridis et al., 1999)), and execution data based approaches (e.g., (Qin et al., 2009), (Allier et al., 2009), (Allier et al., 2010)). A common drawback of many

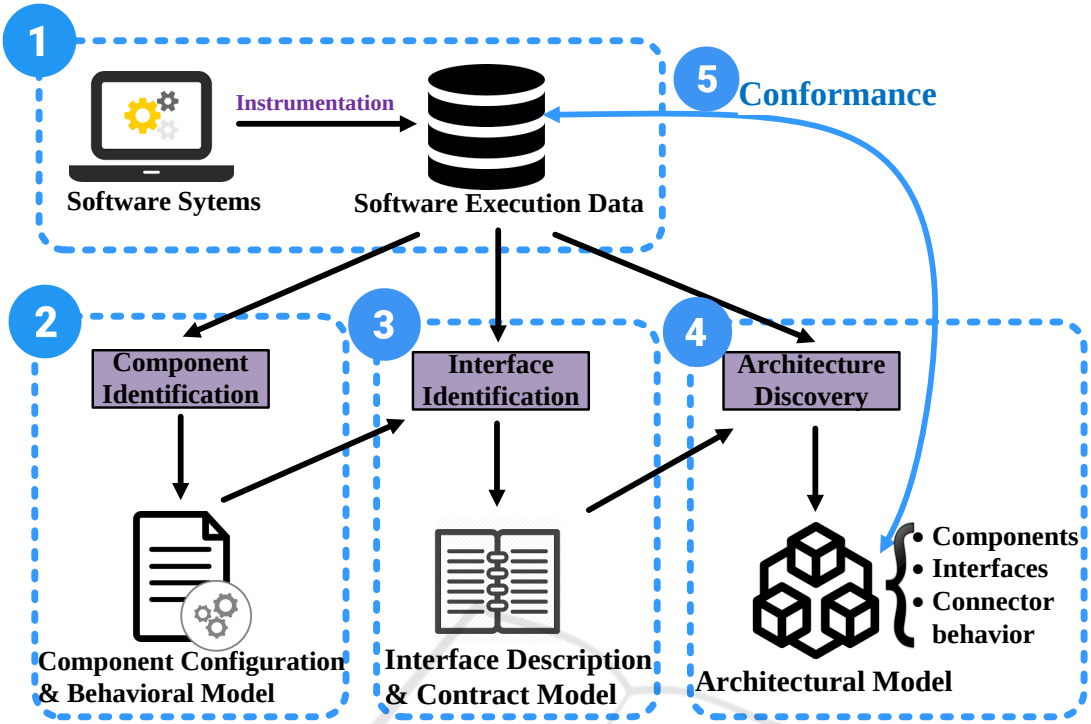

Figure 1: Research Overview.

of these approaches is the lack of tool support which hinders their applicability.

As for interface identification, *Simon et al.* (Allier et al., 2011) propose to identify interfaces by grouping methods of the same class, i.e., one interface for each class if this class has methods that are used by other components. Hence, this approach defines interfaces based on the internal structure of components. *Seriai et al.* (Seriai et al., 2014) give an approach to identify interfaces of a component by grouping methods that are called by another component. Methods that are used by the same component(s) are grouped as an interface. Hence, this approach leads to different components using a single interface regardless of the functions they need. With respect to architectural model reconstruction, *Simon et al.* (Allier et al., 2011) propose to discover an architectural model from source code to help understand the behavior of a software system. The components are extracted as a set of classes and interfaces are identified by grouping methods of same classes and the interactions among components are represented by binding interfaces in a static way. Differently, *Seriai et al.* (Seriai et al., 2014) also presents an approach to discover architectural model from source code whereas interfaces of a component are identified by grouping methods that are called by the same component.

By taking execution data as input, Dragomir and Lichter (Dragomir and Lichter, 2013) try to present an architectural description by visualizing object-level interactions based on sequence diagrams. However, object-level information are too fine-grained and not understandable for large-scale software. In addition, the interactions among components are also represented by simply binding interfaces in a static manner, which neglects the behavioral aspects.

4 METHODOLOGY

Figure 1 gives an overview of our methodology, based on which we describe the approaches adopted in the research. Generally speaking, we start from the instrumentation and standardization of software execution data (see ① in Fig. 1). By taking the standardized execution data as input, we identify components and discover component behavioral models as shown in Fig. 1 ②. Then, we identify interfaces and discover interface contract models for each component as shown in Fig. 1 ③. Next, the architectural model can be discovered as shown in Fig. 1 ④. Finally, we evaluate the conformance between the architectural model and the execution data (see ⑤ in Fig. 1).

4.1 Standardization of Software Execution Data

The input of our research is software execution data, which can be obtained by instrumenting and monitoring real software execution. The software execution data consist of method calls. Normally, a method call records software-specific information, including the method name, the class name, the object that invokes this method, the package name, the line number of the method, the input parameter types and values of the method, the start time (in nanosecond precision), complete time (in nanosecond precision), the caller method name, the caller class name, the caller object, the caller package name and etc.

To the best of our knowledge, there is no standardization of software execution data which supports reproducibility and shareability of existing research results. The XES standard (Verbeek et al., 2011) defines a grammar for a tag-based language whose aim is to provide designers of information systems with a unified and extensible methodology for capturing systems behaviors by means of event logs and event streams. It is supported by XES Working Group[1]. To provide a unified format for software execution data exchange, we introduce a standardized XES-based extension, i.e., *Software Event Extension*.

The software extension defines the called class name, the called package name, the line number of the called method, the input parameter types and values of the called method, the caller method name, the caller class name, the caller object, the caller package name, and the timestamp for software events within a log. For more detailed information, please refer to (Leemans and Liu, 2017).

4.2 Component Identification and Behavioral Model Discovery

Generally speaking, the identification of components from software execution data is based on clustering classes (Liu et al., 2018a). To understand the behavior of each component, we propose to discover a behavioral model per component using process mining techniques.

- **Component Identification.**
  - ***Step 1: Class Interaction Graph Construction.*** Starting from the software execution data, we propose to construct a class interaction graph (CIG). In the CIG, each node represents a class and each edge represents the calling relation among the connected two classes.
  - ***Step 2: Component Identification from Class Interaction Graph.*** By taking the constructed CIG as input, we partition it into a set of subgraphs using community detection algorithms (e.g., Newman's algorithm (Newman, 2006)). Classes that are grouped in the same cluster naturally form a component, known as component configuration.
  - ***Step 3: Quality Evaluation of the Identified Components.*** After identifying a set of components, we want to evaluate the quality of the identified components. Several quality metrics, e.g., size, cohension, coupling, and modularity, will be evaluated.
- **Component Behavioral Model Discovery.**
  - ***Step 1: Component Instance Identification.*** Starting from the original software execution data, we first propose a novel approach to identify component instance. It serves as the basic *case* notion to generate a software event log for each component. Here, a component instance refers to one independent run of a software component.
  - ***Step 2: Hierarchical Software Event Log Construction.*** Because a software (component) usually has a hierarchical structure represented as multi-level nested method calls, the discovered behavioral model should depict this hierarchy nature. For each component, we recursively transform its event log to a hierarchical one using calling relations among methods.
  - ***Step 3: Component Behavioral Model Discovery using Process Mining.*** For each component, we discover a hierarchical behavioral model from its corresponding hierarchial software event log. Given the hierarchy of a software event log, we only need to traverse through different levels of the log and discover a process model for each sub-log. Note that we can use any existing process discovery approach (e.g., Inductive Miner (Leemans et al., 2013)) in this step.

4.3 Interface Identification and Contract Model Discovery

Starting from the software execution data and component configuration, we propose to first identify a set of interfaces for each component by clustering its methods (Liu et al., 2018b). Normally, when an interface is used by a component, the execution of its methods should follow a specific contract. This contract defines the behavior of the interface by explic-

[1] http://www.win.tue.nl/ieeetfpm/doku.php

itly specifying in which order the methods should be invoked. Therefore, for each identified interface, we then discover a behavioral model to represent the actual behavior using process mining techniques.

- **Component Interface Identification.**
  - ***Step 1: Candidate Interface Identification.*** For each component, we identify a set of candidate interfaces by grouping methods with respect to their caller methods. Note that identified candidate interfaces may have duplication problem, i.e., some methods may be included in different interfaces.
  - ***Step 2: Similar Interface Candidate Merge.*** To solve the method duplication problem among candidate interfaces within the same component, we merge similar candidates in a way such that the overlap of shared methods among interfaces is limited to a reasonable range.
  - ***Step 3: Quality Evaluation of the Identified Interfaces.*** After identifying interfaces of each component, we want to evaluate the functional consistency of each interface.
- **Interface Contract Model Discovery.**
  - ***Step 1: Interface Event Log Construction.*** To enable the discovery of interface contract model using process mining techniques, we obtain the event log from the software execution data for each identified interface.
  - ***Step 2: Interface Contract Model Discovery using Process Mining.*** For each interface, we discover a contract model using existing process mining techniques (e.g., Inductive Miner (Leemans et al., 2013)).

4.4 Formal Specification and Discovery of Software Architectural Model

After identifying components and interfaces (as well as their behavioral models), we then try to recover the architectural model of software. The architectural model is composed of components, interfaces and interactions. A component can interact with other components by interaction methods (or interfaces). Each interaction is described by an interaction model that contains a connector behavioral model (a process model describing the behavior of the invoked interfaces) and the interface instance cardinality information. It can be discovered by performing the following steps:

- **Interaction Method Identification.** An interaction method is a method of an interface that can invoke methods (or interfaces) of other components. It can be detected directly from the software execution data.
- **Connector Behavioral Discovery.** For each interaction method, we first generate its interaction log where each event refers to an interface. A connector behavioral model can be discovered from this interaction log.
- **Interface Instance Cardinality Identification.** Interface instance cardinality information reveals the instance level relationships between the interaction method and the invoked interfaces. The cardinality information for each interface can be obtained by investigating the number of interface instances that is invoked by the interaction method.
- **Multi-view Architectural Models.** Besides a detailed architectural view with interface protocol model, cardinality information and connector behavioral model, we provide multiple views to allow users navigate from fined-grained architectural models to coarse-grained ones.

4.5 Conformance Checking based Architectural Model Quality Evaluation

In this section, we propose to evaluate the quality of the discovered architectural model with respect to the execution data. Conformance checking based quality evaluation measures the fitness of the architectural model against the execution data. It involves the following steps:

- **Mapping Execution Data to Architectural Elements.** Given software execution data and an architectural model, we first create the mapping from method calls in the execution data to architectural elements (e.g., interface, interaction model) in the architectural model.
- **Compute Alignment between Software Execution Data and Architectural Model.** Based on the mapping, we compute the alignment between software execution data and the architectural model.
- **Measure the Fitness Between Software Execution Data and Architectural Model.** Based on the computed alignment between the software execution data and an architectural model, we compute the fitness of the architectural model with respect the execution data. It reveals how well the architectural model fits the execution data.

Table 1: Stage of The Research.

Stage	Period	Description
Stage 1	2015-08 $\sim$ 2016-12	(1) Standardize software execution data; and (2) Component identification and behavioral model discovery.
Stage 2	2017-01 $\sim$ 2018-03	(1) Interface identification and contract behavior discovery; and (2) Formal specification and discovery of architectural model.
Stage 3	2018-04 $\sim$ 2019-07	(1) Conformation checking based architectural model evaluation; (2) Empirical evaluation using real-life software systems; and (3) Finish the Ph.D thesis.

4.6 Empirical Evaluation using Real-life Software Systems

Based on open-source software systems and their execution data (e.g., *JUnit 3.7*[2], *JGraphx*[3], *JHotdraw*[4]), we perform a comprehensive empirical evaluation of all proposed approaches. In addition, we also plan to contribute some real-life case studies where the feedback from stateholders are available. The evaluation should involve the following aspects:

- For component identification, we evaluate the cohesion and coupling metrics for different community detection or graph clustering algorithms (e.g., (Qin et al., 2009), (Allier et al., 2009), (Allier et al., 2010)).
- For interface identification, we compare our approach with existing interface identification approaches (e.g., (Allier et al., 2011) (Seriai et al., 2014)).
- For architectural model discovery, we evaluate our approach that combining different component/interface identification strategies. In addition, we also compare our discovered architectural model with existing ones if possible (e.g., (Dragomir and Lichter, 2013)).

5 EXPECTED OUTCOME

This section describes in detail the expected outcome of our research. It includes the following:

- An XES-based software extension to support the standardization of software execution data.
- An extensible framework to support the identification of components from software execution data. This framework should implement various community and clustering algorithms.
- A process mining based approach to discover hierarchical behavioral models for each identified component.
- An extensible framework to support the interface identification. This framework should implement various of interface identification strategies.
- A process mining based approach to discover contract models for each identified interface.
- An effective approach to support the discovery of multi-view architectural models from software execution.
- An effective approach to support conformance checking between the discovered architectural model and software execution Data.
- A set of user-friendly software tools that support the previous techniques.

6 STAGE OF THE RESEARCH

This research is fully supported by the *NIRICT 3TU.BSR (Big Software on the Run)* research project[5]. This high-profile project is a collaboration between 3 universities, 6 research groups. It starts from August 1, 2015 and runs four years until July, 2019. My role in the project is mainly on creating more abstract representations of the massive amounts of software event data. We aim to develop techniques for generating models and visualizations showing what is really going on in a software system or collection of systems. Generally, we organize the whole research into three stages, as shown in Table 4.4.

REFERENCES

Allier, S., Sadou, S., Sahraoui, H., and Fleurquin, R. (2011). From object-oriented applications to component-oriented applications via component-oriented architecture. In *9th Working IEEE/IFIP Conference*

[2] http://essere.disco.unimib.it/svn/DPB/JUnit%20v3.7/

[3] https://github.com/jgraph/jgraphx

[4] http://www.inf.fu-berlin.de/lehre/WS99/java/swing/JHotDraw5.1/

[5] http://www.3tu-bsr.nl/doku.php?id=start

on Software Architecture (WICSA), pages 214–223. IEEE.

Allier, S., Sahraoui, H., Sadou, S., and Vaucher, S. (2010). Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. *Component-Based Software Engineering*, pages 216–231.

Allier, S., Sahraoui, H. A., and Sadou, S. (2009). Identifying components in object-oriented programs using dynamic analysis and clustering. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 136–148. IBM Corp.

Biermann, A. W. and Feldman, J. A. (1972). On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, (6):592–597.

Briand, L. C., Labiche, Y., and Leduc, J. (2006). Toward the reverse engineering of uml sequence diagrams for distributed java software. *Software Engineering, IEEE Transactions on*, 32(9):642–663.

Chang, S. H., Han, M. J., and Kim, S. D. (2005). A tool to automate component clustering and identification. In *International Conference on Fundamental Approaches to Software Engineering*, pages 141–144. Springer.

Chiricota, Y., Jourdan, F., and Melançon, G. (2003). Software components capture using graph clustering. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 217–226. IEEE.

Cui, J. F. and Chae, H. S. (2011). Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems. *Information and Software technology*, 53(6):601–614.

Dragomir, A. and Lichter, H. (2013). Run-time monitoring and real-time visualization of software architectures. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 1, pages 396–403. IEEE.

Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101. ACM.

Hasheminejad, S. M. H. and Jalili, S. (2015). Ccic: Clustering analysis classes to identify software components. *Information and Software Technology*, 57:329–351.

Ivica, C., Severine, S., Aneta, V., and Michel, C. (2011). A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615.

Kebir, S., Seriai, A.-D., Chaoui, A., and Chardigny, S. (2012a). Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, pages 1–8. ACM.

Kebir, S., Seriai, A.-D., Chardigny, S., and Chaoui, A. (2012b). Quality-centric approach for software component identification from object-oriented code. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 181–190. IEEE.

Kim, S. D. and Chang, S. H. (2004). A systematic method to identify software components. In *11th Asia-Pacific Software Engineering Conference, 2004.*, pages 538–545. IEEE.

Lee, J. K., Jung, S. J., Kim, S. D., Jang, W. H., and Ham, D. H. (2001). Component identification method with coupling and cohesion. In *Eighth Asia-Pacific Software Engineering Conference, 2001. APSEC 2001.*, pages 79–86. IEEE.

Leemans, M. and Liu, C. (2017). Xes software event extension. *XES Working Group*, pages 1–11.

Leemans, M. and van der Aalst, W. (2015). Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In *18th International Conference on Model Driven Engineering Languages and Systems*, pages 44–53. IEEE.

Leemans, S. J., Fahland, D., and van der Aalst, W. (2013). Discovering block-structured process models from event logs-a constructive approach. In *Application and Theory of Petri Nets and Concurrency*, pages 311–329. Springer.

Liu, C., van Dongen, B., Assy, N., and van der Aalst, W. (2016). Component behavior discovery from software execution data. In *International Conference on Computational Intelligence and Data Mining*, pages 1–8. IEEE.

Liu, C., van Dongen, B., Assy, N., and van der Aalst, W. (2018a). Component identification from software execution data: An approach based on newman's spectral algorithm. In *International Conference on Program Comprehension*, pages 1–4, under review. ACM.

Liu, C., van Dongen, B., Assy, N., and van der Aalst, W. (2018b). Component interface identification and behavioral model discovery from software execution data. In *International Conference on Program Comprehension*, pages 1–10, under review. ACM.

Liu, C., van Dongen, B., Assy, N., and van der Aalst, W. (2018c). A framework to support behavioral design pattern detection from software execution data. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 1–12.

Liu, C., van Dongen, B., Assy, N., and van der Aalst, W. (2018d). A general framework to detect behavioral design patterns. In *40th International Conference on Software Engineering*, pages 1–2, accepted.

Lo, D., Mariani, L., and Pezzè, M. (2009). Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 345–354. ACM.

Luo, J., Jiang, R., Zhang, L., Mei, H., and Sun, J. (2004). An experimental study of two graph analysis based component capture methods for object-oriented systems. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 390–398. IEEE.

Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 50–59. IEEE.

McGavin, M., Wright, T., and Marshall, S. (2006). Visualisations of execution traces (vet): an interactive plugin-based visualisation tool. In *Proceedings of the 7th Australasian User interface conference-Volume 50*, pages 153–160. Australian Computer Society, Inc.

Newman, M. E. (2006). Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582.

Qin, S., Yin, B.-B., and Cai, K.-Y. (2009). Mining components with software execution data. In *International Conference Software Engineering Research and Practice.*, pages 643–649. IEEE.

Rubin, V., Günther, C., van der Aalst, W., Kindler, E., van Dongen, B., and Schäfer, W. (2007). Process mining framework for software processes. In *Software Process Dynamics and Agility*, pages 169–181. Springer.

Rubin, V., Lomazova, I., and van der Aalst, W. (2014). Agile development with software process mining. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 70–74. ACM.

Seriai, A., Sadou, S., Sahraoui, H., and Hamza, S. (2014). Deriving component interfaces after a restructuring of a legacy system. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 31–40. IEEE.

Stéphane, D. and Damien, P. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591.

van der Aalst, W. (2015). Big software on the run: in vivo software analytics based on process mining (keynote). In *Proceedings of the 2015 International Conference on Software and System Process*, pages 1–5. ACM.

van der Aalst, W. (2016). *Process Mining: Data Science in Action*. Springer.

van der Aalst, W., Kalenkova, A., Rubin, V., and Verbeek, E. (2015). Process discovery using localized events. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 287–308. Springer.

Verbeek, H., Buijs, J. C., Van Dongen, B. F., and Van Der Aalst, W. M. (2011). Xes, xesame, and prom 6. In *Information Systems Evolution*, pages 60–75. Springer.

Walkinshaw, N. and Bogdanov, K. (2008). Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE Computer Society.

Washizaki, H. and Fukazawa, Y. (2005). A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer programming*, 56(1-2):99–116.