Middleware Adaptation through Process Mining

Nelson Rosa
Centro de Informática
Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil
Email: nsr@cin.ufpe.br

***Abstract*—The development of adaptive middleware systems is a complex task due to the difficulty of dealing with adaptation issues, such as *how* to implement the adaptation mechanism, *where* to insert the adaptive code into the middleware, and *when* the adaptive code is composed with the middleware logic. Existing solutions to build adaptive middleware usually concentrate on the use of software technologies like aspect oriented programming and computational reflection to face with the *how* issue. In this paper, we propose a solution to build middleware that is adapted at runtime (*when*), whose adaptation decisions and actions are moved from the middleware to an external component (*where*) and whose adaptation makes use of process mining techniques and software architecture (*how*). The adaptation process is triggered based on the verification of the middleware event log. In order to evaluate the proposed approach, we carried an experimental evaluation to check the quality of the mined middleware model and the verification overhead.**

***Keywords*-adaptive middleware; process mining; software architecture.**

The development of an adaptive middleware still being a challenge due to the complexity of dealing with three key aspects already known to build adaptive software [1]: *how* to implement the adaptive code while guarantee middleware safety after successive adaptations; *when* the adaptation must occur, e.g., at development, compile, deployment or runtime; and *where* the adaptive code is inserted, e.g., application, middleware, or external element.

The development of adaptive middleware systems still being a challenge due to the complexity of deal with the aforementioned reconfiguration issues. Meanwhile, this is not a recent topic in the middleware community [2][3][4]. More recently, adaptive middleware systems have been developed in several different application domains, such as Internet of Things [5] and cloud computing [6]. However, whatever the approach or application domain, they mainly focus on adopting or extending existing enabling technologies to solve the adaptation issues.

In this context, we adopt emerging process mining techniques as an enabling technology for middleware adaptation (*how*). Process mining is a powerful way to analyse software's operational aspects based on event logs generated at runtime [7]. It enable us to verify behavioural and performance properties of the software while it is in execution, and correlate distribute events.

In this paper, we present an extension of MIstRAL [8], namely μ-MIstRAL (mining-based extension of MIddleware Reconfiguration Aid by formaLism), that works as an end-to-end solution to build adaptive middleware systems based on the use of process mining and software architecture principles, i.e., it helps in developing, executing and adapting middleware systems. The novelties in μ-MIstRAL include the use of process mining for middleware process discovery (development) and triggering the middleware adaptation (runtime), and the adoption of software architecture principles for describing the middleware behaviour and structure. More specifically, we use process mining techniques to check middleware's behavioural properties and then trigger the adaptation if the property does not hold. In practice, given a middleware execution log, check if this log has some deviation from the middleware expected behaviour. If a deviation exists, then the middleware is reconfigurated to fix it.

Benefits of adopting processing mining include: middleware mining allows the application mining, middleware mining leads to application insights, mining gathers information about what is actually happening (log) and not what developers think that is defined as a model, and possibility of checking performance properties and generating the middleware models. Further minor advances of μ-MIstRAL include the automatic code instrumentation of the middleware to generate event traces, and the generation of XES (eXtreme Event Stream)-based event logs [9].

This paper is organized as follows. Section I introduces basic concepts of process mining and MIstRAL. Next, Section II presents an approach to put together adaptive middleware and process mining. Section III makes an evaluation of the proposed approach. Section IV presents existing researches on adaptive middleware and runtime verification. Finally, Section V presents conclusions and some future work.

I. Basic concepts

This section presents basic concepts of process mining and the existing version of MIstRAL, which is extended in this paper.

A. *Process Mining*

Process mining uses event logs to gain insights into the process being executed. In practice, the key idea of process mining is to diagnose business processes by mining event logs for knowledge. In turn, an event log consists of a set of traces where each trace corresponds to an execution of the process. A trace contains a sequence of activities executed by the process [10].

Process mining techniques have been widely used with three main purposes: process discovery, conformance checking and performance analysis [7]. Process discovery means to generate a formal software model, e.g., Petri net or process tree specification, under real-life conditions from the event log. The software model describes dynamic, operational, interactive aspects and real-life performance of the software. Meanwhile, the conformance checking searches for detecting deviations between the behaviour recorded in the log and one expressed by a software model. The software model generated from the event log can also be replayed in such way that it is possible to identify bottlenecks, delays and inefficiencies of the software. Finally, the log can also be verified to check desired properties as proposed in [11]. In this case, given an event log and some property, it is possible to verify whether the property holds or not.

Three key activities must be performed for process mining: *Instrumentation* addresses how to obtain information from the system being executed, which involves the instrumentation strategy; *Data Collection* needs to define a data collecting infrastructure able to merge information from different sources; and *Process Discovery*, which is the set of steps necessary to find a model that help us to gain insights into the software.

B. *MIstRAL*

MIstRAL (MIddleware Reconfiguration Aid by formaLism) [8] is the implementation of an adaptation module that uses formal elements, in a lightweight way, to guide the middleware adaptation process. The use of MIstRAL assumes that the middleware's adaptable components are described in a chain of interceptors that defines the middleware's interceptors and the sequence they are executed. MIstRAL requires that the middleware must be able to generate a simple event log, e.g., an ordered sequence of actions performed by the middleware.

Figure 1 shows a general overview on how MIstRAL works in practice. At runtime, the middleware generates an event log of all activities performed by it. The log is read by MIstRAL (1) that forwards it to the CADP Toolbox[1] along with some properties it must satisfy (2). CADP uses a model checker to verify if the properties are actually being satisfied (3). For example, a typical property states that "*action 'a' always occurs after action 'b'*" when the middleware is running properly. If the property is not satisfied, MIstRAL applies an adaptation plan that changes the chain of interceptors and consequently the middleware behaviour (4).

[1]http://cadp.inria.fr

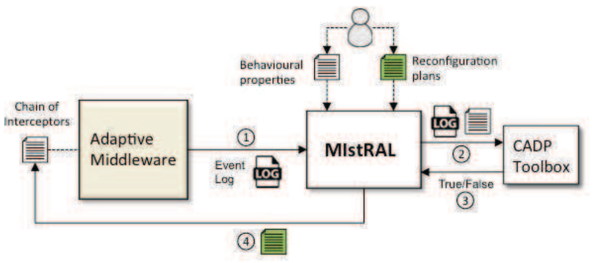


Figure 1. General overview of MIstRAL

II. Middleware process mining

As mentioned before, the proposed extension of MIstRAL , namely μ-MIstRAL, advances in two main aspects: the use of process mining techniques [7], and adoption of software architecture principles to design the midleware. In this way, μ-MIstRAL is an initial end-to-end solution to design adaptive middleware whose design and runtime adaptation is driven by process mining. Figure 2 shows the main elements of μ-MIstRAL.

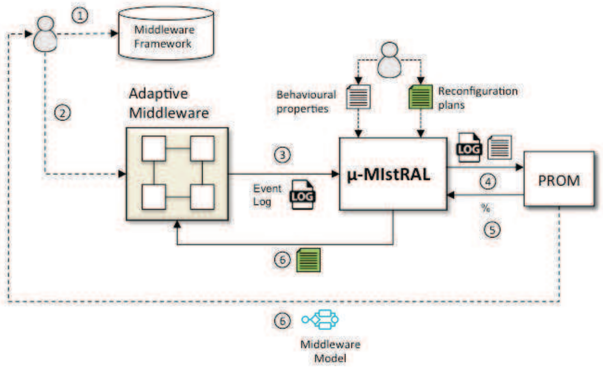


Figure 2. General overview of μ-MIstRAL

This figure shows that developers use the middleware framework to build the adaptive middleware instead of developing it from scratch (1-2). As a consequence, the artifact that is actually adapted in this case is the software architecture instead of a chain of interceptors. Steps 3 and 4 are similar in both versions. However, in μ-MIstRAL, the event log is used to verify the log (online) and mine the middleware model (offline), e.g., a process tree or Petri model that describes the middleware behaviour (6). When used to verify the log at runtime, it triggers the adaptation process if a property does not hold. On the other hand, the mined model can help the middleware developer at development time to conformance checking. Table I shows a summary of differences between MIstRAL (see Section I-B) and μ-MIstRAL.

Table I
MISTRAL VERUS μ-MISTRAL

Aspect	**Mistral**	**μ-MIstRAL**
Code instrumentation	Manual	Automatic
Process discovery	None	Mining algorithms
Event log	Simple log	XES-based log
Verification tool	CADP	ProM [12]
Verifiable properties	Behavioural	Behavioural and performance
Adaptation artifact	Chain of interceptors	Software architecture
Adaptation trigger	LTL formula not satisfied	LTL formula not satisfied

The adoption of processing mining techniques in the middleware domain has also some challenges:

- *Distributed sources*: Being a distributed software, middleware's log is generated from distributed sources that demand an strategy to collect, store and order the events of interest;
- *Application triggered*: Most middleware internal services, mechanisms and processes are triggered by distributed applications built atop it. Hence, while in the traditional use of process mining techniques an user triggers activities of the system, an application has a similar role in the case of middleware;
- *Performance impact*: Being an infrastructure software used to help distributed applications at runtime, the middleware mining can affect both application and middleware performance;
- *Reusable software*: As a highly reusable software, the mining of a middleware process can demand a lot of effort, but it produces a process model that works whatever the application built atop it;
- *Complexity*: Middleware systems are inherently complex as they usually provide a large number of transparencies and services to application developers. As a consequence, the mining process strategy must adopt the principle of *divide and conquer* to deal with this multitude of aspects to be mined. For example, it should be interesting to break up the middleware log considering which middleware service, component or transparency is being exercised.

Meanwhile, the use of software architecture principles has a great impact on the middleware development and adaptation. The development of the middleware is performed by using a provided framework containing components and connectors specially designed to build adaptive middleware. These elements are composed into a configuration that is the artifact to be reconfigured at runtime.

Next sections presents the proposed software architecture framework and all steps of the adoption of process mining.

A. *Software architecture framework*

As mentioned before, the middleware software architecture is the artifact that is reconfigured at runtime when a desired property is not satisfied. By reconfiguring the software architecture, the middleware behaviour is altered in such way that the property still hold.

The software architecture framework consists of a set of software architecture elements, namely *Port*, *Interface*, *Component*, *Connector* and *Configuration* implemented in Java. This structural set of elements is enriched with the inclusion of behavioural descriptions (*Behaviour*) of components and connectors as formally proposed by Wright [13].

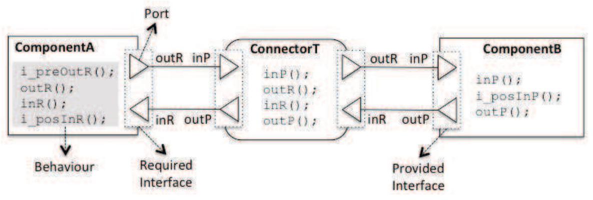


Figure 3. Configuration overview (Development time)

Figure 3 shows a simple *configuration* having two components (*ComponentA* and *ComponentB*) and one connector (*ConnectorT*). Each element has one or more interfaces and its description also includes the set of actions it can engage (*Behaviour*). The behaviour of a component is described by defining the temporal ordering of *internal* and *external* actions performed by the component. Similarly to process algebras [14], internal actions are ones not visible by the external environment and related to the functionality of the component, whilst external actions models the interactions with other elements. For example, *ComponentA* performs internal action *i_preOutR* followed by an external action *outR* to interact with *ConnectorT*. External actions are carried out through *ports*.

Ports are used to communicate component/connectors with their external environment and serve as synchronisation points. For example, *ComponentA* synchronises with *ConnectorT* in ports *outR* and *inR*. Hence, *ComponentA* only executes action *outR* if *ConnectorT* is ready to execute action *inP*. Two kinds of ports have been considered, namely *output* and *input* ports. Output ports are used to send messages to external elements, while input ports are used to receive messages from the external environment. For example, *ComponentB* has one input port (*inP*) and one output port (*outP*). Every port belongs to an *interface*.

Interface is a set of ports grouped logically. Two kinds of interfaces are being considered: *provided* and *required*. Provided interfaces are used by external elements to request operations to a component/connector, whilst a required interface includes ports used by a component/connector to make request and receive response from an external element. In

Figure 3, *ComponentA* has a required interface and *ComponentB* has a provided interface. Meanwhile, *ConnectorT* has provided and required interfaces.

A *Component* is a loci of computation or data storage. Two types of components are considered: *simple* and *composite*. A composite component include two or more simple components connected through connectors. A component has one or more provided and required interfaces.

In addition to the definition of component itself, the framework also includes a set of commonly used components, such as *Client*, *Server*, *ClientServer*, *ClientRemote* and *ServerRemote*. Two particular components are also defined, *OutputWrapper* and *InputWrapper*, used to allow the interaction of architectural elements and objects of object oriented (OO) applications. In this way, application developers are free to not use software architecture concepts to build their applications, i.e., the application can be OO.

A *Connector* is responsible to communicate components. As a basic rule in software architecture, a connector must exist between any two components. One type of connector is provided, namely *RequestReply*, which allows synchronous communication between client and server components. As shown in Figure 3, *ConnectorT* explicitly defines how *ComponentA* and *ComponentB* interact, i.e., the only possible interactions are: *ComponentA* sends a message to *ComponenB* followed by *ComponentB* sends a message to *ComponenA*.

By using the proposed software architecture framework, we provide a set of built-in middleware components and connectors specially designed for implementing middleware. These built-in components and connectors have been inspired on the Remoting Patterns [15] that includes thirty one architectural patterns for implementing RPC-based middlweare systems. Each middleware component was implemented by reusing a component of the software architecture framework and then defining the internal actions that implement the component's functionality.

The middleware framework includes the following components with their respective basic software architecture components: *Requestor* (extends *Component*), *ClientProxy* (extends *InputWrapper*), *Invoker* (extends *Component*), *Client Request Handler* (extends *ClientRemote*), *Server Request Handler* (extends *ServerRemote*), *Marhsaller* (extends *Server*), *Lookup* (extends *Server*), *Invocation Interceptor* (extends *Server*), *Lifecycle Manager* (extends *ClientServer*), *QoS Observer* (extends *Server*), and *Location Forwarder* (extends *ClientServer*). An unique connector has been defined, namely *Sync* (extends *RequestReply*). This connector allows synchronous communication between components based on a request-reply interaction, i.e., a component makes a request to another component and waits for a reply.

B. Code instrumentation

As needed for the middleware process mining, the code instrumentation consists of injecting into the source or binary code the needed code to produce the middleware event log. This process can be manual or automatic. It is worth observing that any injection like this is dependent of the implementation language, needs to have a minimum impact on system's performance, (ideally) does not impose any additional effort to application developers, and need to generate enough information to the conformance checking and verification. Meanwhile, the instrumentation obviously should not change the functionality of the code being instrumented.

Some techniques can be used in the instrumentation phase, such as binary weaving and code transformation. In particular, we adopted the code transformation that automatically executes this task in the middleware source code. As we adopted software architecture principles, the instrumentation is performed on the code of components and connectors that make up the middleware.

The instrumentation consists of injecting code into the operations provided by all middleware components and connectors. The injected code generates *start* and *complete* events when the operation is invoked and completed, respectively.

C. Data collection

After instrumenting the middleware code, it generates traces which are grouped into the middleware event log. This task consists of collecting data from distributed sources (middleware components) and then send it to a central point responsible for consolidating all events in a single log.

As expected, a single middleware trace has events generated by middleware components running on different computation nodes. This fact has an impact on the time of occurrence of middleware events. For example, a remote request from application's component a_1 to a_2 generates traces with events from hosts $host_1$ and $host_2$ whose clock time should be synchronised. Collected data should be asynchronously sent to the middleware event log for performance reasons.

D. Process Discovery

Process discovery consists of mining the middleware event log and then produces a model that can be used to express the actual behaviour of the middleware, i.e., the model of what is actually executing. Currently, ProM generates models expressed in Petri net, process tree, and BPMN.

As we have adopted ProM for the processing mining, it already provides a set of mining algorithms, such as inductive miner, fuzzy miner, Petri net miner, multiperspective miner, heuristics miner and BPMN miner. These algorithms can be used to mine the whole middleware or a particular middleware service, mechanism or protocol. In the case the interest is on these specific elements, filters can be used to remove events strange to the component being mined, e.g., only consider client-side middleware events, naming service interactions, or event with highest frequency in the middleware. A large number of ProM filters are made available and can be combined in several different ways to help in yielding very confident software models.

At this point, there are some basic facts about this process discovery that must be clarified: the process discovery should be (preferably) performed offline, filters and miner algorithms can be used repeatedly several times before a good process model can be found, the obtained process model can be or not used in the conformance checking.

Finally, a good process model allows for most of the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end [16].

E. Conformance Checking

As mentioned in Section I-A, the conformance checking looks for deviations between the middleware's behaviour recorded in the event log and one expressed by the middleware model. It is worth observing that the middleware model can be created by hand or automatically generated as described in the previous section. In particular, we are only considering those models automatically generated by ProM.

In practice, the conformance checking needs for an alignment of event log and process model, i.e., events in the event log need to be related to model elements and vice versa. Such alignment shows how the event log can be replayed on the process model [16].

Figure 4 depicts the process model of a middleware's naming service along with its path deviations generated from an event log. This model gives insights on the activities (blue ellipses) performed by the naming service along with the frequency they happen (numbers inside the ellipse), e.g., *lookup* and *bind* activities were executed just one time. Deviations show precisely what parts of the model deviate with respect to the log.

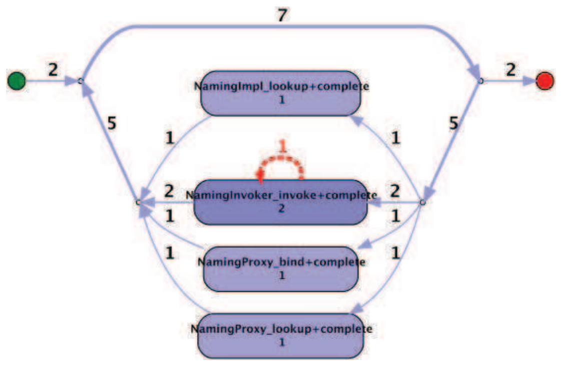

7
2
2
5
5
NamingImpl_lookup+complete
1
NamingInvoker_invoke+complete
2
NamingProxy_bind+complete
1
NamingProxy_lookup+complete
1

Figure 4. Paths and deviations of a process model

The edge circumventing event *NamingInvoker_invoke+complete* (dashed edge) indicates the existence of an event in the log that is not in the model. As mentioned before, a model with good fitness allows for most of the behavior seen in the event log, whilst a model having a perfect fitness allows all traces be replayed by the model from beginning to end.

F. Verification

The verification consists of checking a temporal logic property on the middleware event log. The result of this verification defines whether the property is satisfied or not. In particular, the verification is performed using the LTL checker and the properties are specified in a Linear Temporal Logic-based language [11].

The properties adopted in this paper follows the set of property patterns defined in [17] and supported by ProM: a given event does not occur within a scope (*absence*); a given event must occur within a scope (*existence*); a given event must occur *k* times within a scope (*bounded existence*); a given event occurs throughout a scope (*universality*); an event *a* must always be preceded by an event *b* (*precedence*); an event *a* must always be preceded by an event *b* (*response*); event sequence $e_1, ..., e_n$ must always be preceded by event sequence $f_1, ..., f_m$ (*chain precedence*); and sequence of events $e_1, ..., e_n$ must always be followed by a sequence of actions $f_1, ..., f_m$ (*chain of response*).

G. Adaptation

The adaptation consists of changing the middleware behaviour as the verification identified that something is not working as planned in the middleware. Hence, an intervention is necessary to be performed automatically. The intervention is triggered when the verification of a temporal property on the middleware event log produces a false result. As a consequence, a reconfiguration operation must be executed to change the way the the dynamic components are connected in the software architecture.

In particular, four operations can be performed: add a new component, remove an existing component, replace an existing component or reconnect existing components.

H. Implementation

μ-MIstRAL was implemented in Java and currently uses the ProM to check the middleware event log and mine the middleware formal model. Figure 5 presents a general overview of the proposed implementation. The adaptive middleware is implemented in Java, according to the remoting patterns [15] and whose implementation is a collection of components, connectors and a configuration. For example, the architecture includes components such as *ClientProxy*, *Invokers*, *Marshaller*, and connectors like *RequestReply*.

Middleware *Event Log* is obtained by instrumenting the middleware with a logger that takes responsibility of logging actions executed by the middleware and sends them to *LogServer*. From time to time, *Monitor* invokes *Checker* and uses formal adapters (*CADP Adapter* or *ProM Adapter*)

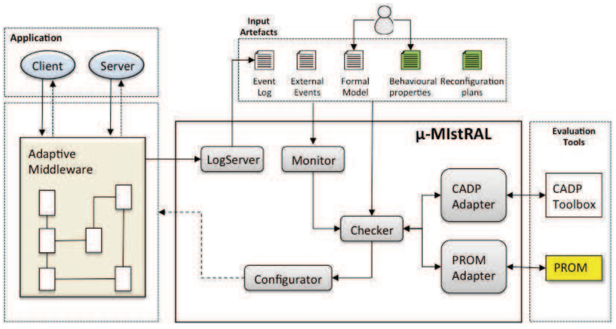


Figure 5. μ-MIstRAL Architecture

to both formatting the log and invoking the model checker according to *Evaluation Tool* being used. *Evaluation Tool* is responsible for the *Conformance Checking*.

The response from *Evaluation Tool* is analysed by the *Checker*. If a reconfiguration is necessary, *Checker* asks for the *Configurator* to perform the reconfiguration plan according to the property that was not satisfied. The reconfiguration plans change the middleware's software architecture, e.g., by replacing a component.

CADP and ProM were adopted as evaluation tools, but other tools like FDR3[2] should also be used. By using ProM, it is possible to check the properties aforementioned, whilst the use of CADP allows to check properties as shown in [8].

III. Experimental Evaluation

The objectives of this evaluation are (i) to assess the quality of the mined middleware model and (2) to evaluate the time spent to perform the verification that triggers the adaptation. The first evaluation is important because it helps us to assess how good is the quality of the middleware event log, whilst the second one gives an idea on the needed time to runtime checking using process mining tools.

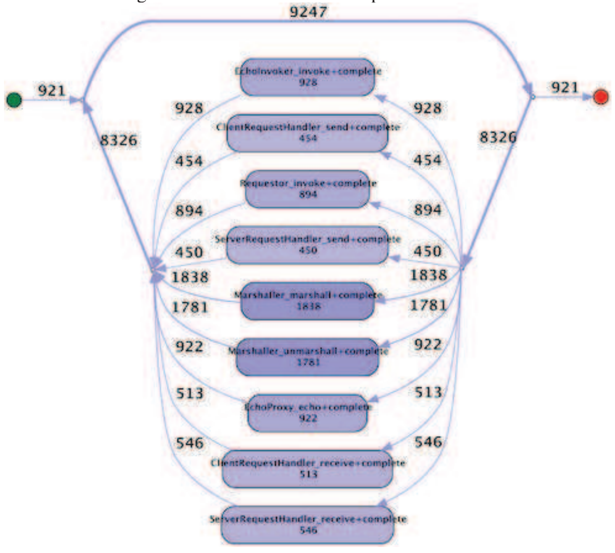
A. *Process Discovery*

An important criterion for the quality of a mined model is the consistency between the mined model and the traces in the event log. Therefore, a standard check for a mined model is to try an execute all traces of the log in the discovered model. If the trace of a case cannot be executed in mined model, there is a discrepancy between the log and the model.

In order to check the quality of the middleware mined model, we generated a middleware log by running a distributed "Echo" application made up of a client and a remote object developed atop the adaptive middleware shown in Figure 5. The client makes 1000 invocations to the remote object, which generate an event log having 18060 events distributed into 01 trace to register the "Echo" service implemented by the remote object into the middleware's naming service; 01 trace to lookup the middleware's naming service for the "Echo" service; and 921 traces corresponding to the invocations to the remote echo. Next, the middleware log was filtered to remove low frequency traces related to the interaction with the naming service, i.e., two traces were removed from the log. Finally, the filtered log was mined using the Inductive Visual Miner [18] and generated the model process shown in Figure 6.

[2]https://www.cs.ox.ac.uk/projects/fdr/

Figure 6. Mined middleware process model


9247
921
8326
928
454
894
450
1838
1781
922
513
546
EchoInvoker_invoke+complete 928
ClientRequestHandler_send+complete 454
Requestor_invoke+complete 894
ServerRequestHandler_send+complete 450
Marshaller_marshall+complete 1838
Marshaller_unmarshall+complete 1781
EchoProxy_echo+complete 922
ClientRequestHandler_receive+complete 513
ServerRequestHandler_receive+complete 546

As show in this figure, the mined model has not dashed edges (see Figure 4), which means that it has a perfect fitness and all 921 traces in the log can be replayed by the model from beginning to end. It is also possible to generate the corresponding Petri net middeware model as depicted in Figure 7.

B. *Verification*

The second evaluation consisted of measuring the time to verify a property on the middleware event log. Figure 8 shows the time spent to check property '<>(activity==α_1) $\wedge$ <>(activity==α_2)'. The number of events refer to ones generated when the client invokes the remote object 1, 10, 100, 500, 1000, 2500, 5000 and 10000 times. For example, 10000 invocations to the remote object produces 178685 events in the event log. The business time of the echo remote object was set to 100ms.

As expected, larger event logs demand more time to be checked. However, even for large number of events, the verification time is short when compared to the total execution time. For example, as the business time was 100ms, 10000 invocations to the remote object lasts about 100 seconds, whilst the verification takes only 6,593 seconds.

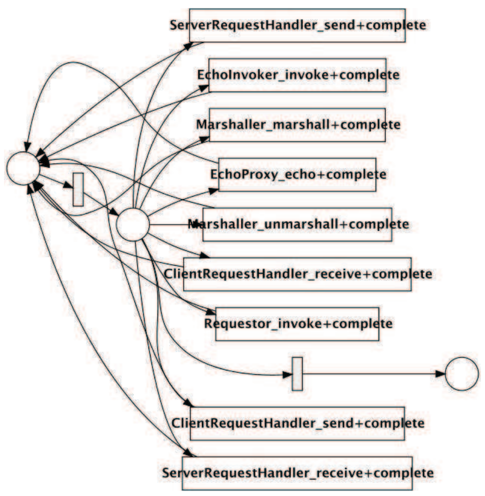

ServerRequestHandler_send+complete
EchoInvoker_invoke+complete
Marshaller_marshall+complete
EchoProxy_echo+complete
Marshaller_unmarshall+complete
ClientRequestHandler_receive+complete
Requestor_invoke+complete
ClientRequestHandler_send+complete
ServerRequestHandler_receive+complete

Figure 7. Petri net mined middleware model

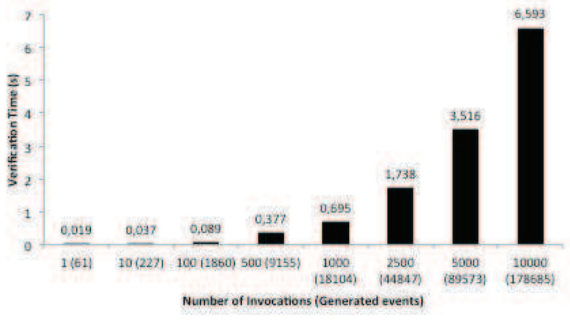


Figure 8. Time to verify the middleware event log

IV. Related Work

Related works about what is being proposed in this paper may be organized into two main categories: enabling techniques of adaptive middleware and runtime analysis of distributed systems.

A. Adaptive Middleware

The design and implementation of adaptive middleware is not a recent topic in the middleware community. Several adaptive middleware systems have been proposed and implemented since a long time ago. Pioneer examples include the reflective middleware DynamicTAO [2], OpenORB [3] and OpenCom [4]. More recently, adaptive middleware have been built in several different application domains, such as wireless sensor networks, cyber-physical systems, multi-tenant applications, large-scale power systems, onboard sattellite systems and public transit system. However, whatever the approach or application domain, most approaches focus on adopting (or even extend) an existing enabling technology (e.g., computational reflection) as the key element to solve the reconfiguration issues. Formal methods are not used in any stage of the aforementioned middleware development and execution.

Attempts to put together FDTs and non-adaptive middleware are also not recent. Formal techniques have been used in different phases of the development of middleware-based applications and middleware development itself. It possible to observe the use of formal description techniques in two different ways: in all phases of middleware development (minority) or in just one phase (majority). The use of formal description techniques in all phases of the middleware development is rare, but it has been already done using SDL (Specification and Description Language). Meanwhile, the use of formalisms in individual phases is widely found and it starts at the elicitation requirements phase. Most common, however, is the adoption of FDTs associated to architectural aspects of the middleware and at design phase. Finally, there are some works on formalising the implementation phase. However, despite the large number of works on middleware formalisation, none of the aforementioned approaches treat with practical (lightweight) aspects of the use of formal methods at runtime.

B. Dynamic Analysis

Dynamic system analysis are used to understand the behaviour of systems at runtime. Dynamic analysis of distributed systems are usually adopted for debugging, maintenance, detect anomalies, identify performance bugs, mine temporal properties, and runtime verification [19][7].

Beschastnikh [19] developed a tool, namely CSight, to infer a model of concurrent system's behaviour by mining event log execution. By using the model, software engineers can understand complex behaviour, detect anomalies, debug, and increase confidence in the implementation correctness. Kumar [20] presents a class level specification mining framework for distributed systems. This approach infer the system specification by running the program and finding patterns in its execution traces that can be interpreted as its specification. Leemans [7] uses process mining techniques to produce formal models of distributed systems. To do that, he defines a methodology composed of three key steps, namely instrumentation, gathering data and discovery business transactions.

A pioneer work on understanding the dynamic behaviour of distributed systems is [21]. Moe defines a set of steps for this purpose: tracing, analysis, presentation and manual adaptation. The tracing is performed by instrumenting the middleware, while the analysis consists of sequence recognition, trace visualisation and statistics calculation. Finally, the trace analysis allows developers and testers to discover some facts about the system and carry out needed improvements.

Finally, it is worth observing that none of these works on dynamic analysis focus on the use of processing mining techniques in the middleware domain.

V. Conclusion and Future Work

This paper presented μ-MIstRAL, an end-to-end approach for building adaptive middleware systems based on the use of process mining techniques and software architecture. μ-MIstRAL uses process mining as the enabling software mechanism to decide when the middleware adaptation must occurs, and adopts software architecture to design the adaptive middleware. Meanwhile, μ-MIstRAL does not insert the adaptation mechanism into the middleware as the mechanism is moved to an external component that decides and takes needed actions to reconfigure the middleware's software architecture.

Our unique contributions in this paper are the use of process mining to trigger the middleware adaptation and the proposition of a middleware software architecture framework for building adaptive middleware. Further minor contributions are related to the actual implementation of an adaptive middleware in Java that uses the proposed approach and the automatic instrumentation of the middleware source code to generate the event log.

We are now using the same approach to implement an adaptive publish/subscribe middleware, by adding new middleware components and connectors, and by extending the set of properties and configuration plans needed to reflect the particularities of this middleware model, such as delivery guarantee and extensive use of queues.

References

[1] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[2] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Middleware 2000*. Springer Berlin / Heidelberg, 2000, vol. 1795, pp. 121–143.

[3] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski, "The Design and Implementation of Open ORB 2," *IEEE Distributed Systems Online*, vol. 2, pp. –, Jun. 2001.

[4] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware," in *Middleware 2001*. Springer Berlin / Heidelberg, 2001, vol. 2218, pp. 160–178.

[5] S. Park and J. Song, "Self-adaptive middleware framework for internet of things," in *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, Oct 2015, pp. 81–82.

[6] D. Van Landuyt, S. Walraven, and W. Joosen, "Variability Middleware for Multi-tenant SaaS Applications: A Research Roadmap for Service Lines," in *SPLC'15*, ser. SPLC '15. New York, NY, USA: ACM, 2015, pp. 211–215.

[7] M. Leemans and W. van der Aalst, "Process mining in software systems: Discovering real-life business transactions and process models from distributed systems," in *MODELS'15*, Sept 2015, pp. 44–53.

[8] N. Rosa, "Middleware Reconfiguration Relying on Formal Methods," in *2015 IEEE International Conference on Computer and Information Technology*, Oct 2015, pp. 648–655.

[9] IEEE, *Draft Standard for XES - eXtensible Event Stream*, IEEE Computational Intelligence Society Std., June 2016.

[10] W. van der Aalst, *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, 2011.

[11] W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen, *Process Mining and Verification of Properties: An Approach Based on Temporal Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 130–147.

[12] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, *XES, XESame, and ProM 6*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 60–75.

[13] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 3, pp. 213–249, Jul. 1997.

[14] W. Fokkink, *Introduction to Process Algebra*, G. R. W. Brauer and A. Salomaa, Eds. Springer, 2000.

[15] M. Volter, M. Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet and Real Time Distributed Object Middleware*. John Wiley & Sons Ltd, 2005.

[16] W. van der Aalst, A. Adriansyah, and B. van Dongen, "Replaying history on process models for conformance checking and performance analysis," *Wiley Int. Rev. Data Min. and Knowl. Disc.*, vol. 2, no. 2, pp. 182–192, Mar. 2012.

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 411–420.

[18] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Process and Deviation Exploration with Inductive Visual Miner," in *Proceedings of the BPM Demo Sessions 2014 Co-located with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014.*, 2014, p. 46.

[19] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight," in *ICSE'14*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 468–479.

[20] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Inferring Class Level Specifications for Distributed Systems," in *ICSE'12*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 914–924.

[21] J. Moe and K. Sandahl, "Using Execution Trace Data to Improve Distributed Systems," in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 640–648.