2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing

Middleware Reconfiguration Relying on Formal Methods

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

***Abstract*—Adaptive middleware is a software for developing and executing distributed applications and can be reconfigured at runtime without its complete stop. The need of reconfiguration is usually triggered by changes in application's requirements and environmental conditions, to fix bugs or to extend the middleware's functionality. The development of an adaptive middleware still being a challenge due to the complexity of dealing with adaptation issues, such as how, when and where to reconfigure the middleware. Existing solutions to build adaptive middleware concentrate on the use of software mechanisms like aspect oriented programming and computational reflection to face the issues behind the reconfiguration. In this paper, we propose to reconfigure the middleware by moving adaptation decisions and actions from the middleware to an external and formally-based element. The whole adaptation process is performed based on the behavioural analysis of the middleware execution trace. In order to evaluate the proposed approach, we carried experimental experiments to check the effectiveness of the proposed adaptation mechanism and measure the overhead caused by the proposed adaptation mechanism.**

***Keywords*-middleware; adaptation; formal methods.**

I. Introduction

Middleware platforms are widely recognised as complex software systems [1][2]. This complexity comes mainly from the need of providing an increasing number of transparencies and services to distributed application developers. In practice, application developers want to keep away from the difficulty of treating with underlying concurrency, communication, and distribution mechanisms, whilst they also demand distributed services to aggregate values to their application. As a consequence, it is expected that middleware platforms should provide distribution transparencies such as access, location, failure and migration that hide low level distribution mechanisms. At the same time, the roll of available middleware services usually includes security, concurrency, licensing, deployment, and so on.

Adaptive middleware is a particular kind of middleware whose behaviour can be modified at runtime without its complete stop. Fours key aspects are usually considered in the design of adaptive middleware systems [3]: *why* the adaptation is necessary, *when* the adaptation must occur, *how* the adaptation is carried out and *where* the adaptation code is inserted in the middleware. The motivation for middleware adaptation is based on the need of adapting to changes of application's requirements or application's behaviour, adapting to changes of environmental conditions, fixing middleware's bugs or extending the middleware functionality. The time adaptation occurs is usually at development time, compilation time, deployment time and runtime. The adaptation is usually implemented using software enabling mechanisms like computational reflection, and two traditional strategies are usually adopted [3]: parameter adaptation, which modifies the middleware variables that determine its behaviour; or compositional adaptation, which exchanges algorithmic or structural middleware components with others to help the middleware to fit to changes in its environmental conditions. Finally, the place where the adaptive code is usually inserted is variable: middleware layers or application code.

The development of an adaptive middleware still being a challenge due to the complexity of deal with aforementioned reconfiguration issues. Meanwhile, this is not a recent topic in the middleware community [4][5][6]. More recently, adaptive middleware systems have been developed in several different application domains, such as large-scale power systems [7], onboard sattellite systems [8], and public transit system [9]. However, whatever the approach or application domain, these approaches focus on adopting (or even extending) an existing enabling technology (e.g., computational reflection) to solve the adaptation issues.

In this paper, we present an approach, namely MIstRAL (MIddleware Reconfiguration Aid by formaLism), for building adaptive middleware systems based on the lightweight use of formal methods. MIstRAL works at runtime; assumes that the reason for reconfiguration is the existence of bad behaviour in the middleware (e.g., deadlock) or changes in the application requirements; uses formalisms as enabling technology and does not incorporate the reconfiguration activities into the middleware.

Similarly to the aforementioned approaches, MIstRAL also releases middleware developers from adaptation activities. However, MIstRAL advances in two different directions in relation to existing solutions. Firstly, we use a formal approach as our enabling software technology to deal with the middleware adaptation (how). Secondly, we do not insert the adaptation mechanism into the middleware (where). The proposed mechanism is moved to an external component that decides and takes needed actions to reconfigure the middleware.

978-1-5090-0154-5/15 $31.00 © 2015 IEEE
DOI 10.1109/CIT/IUCC/DASC/PICOM.2015.93

IEEE computer society

In practice, MIstRAL helps to identify that something is not working properly in the middleware and then fix its behaviour; and helps to identify that the application is not working properly (e.g., due to the current middleware configuration) and then adapt the middleware to fix the application behaviour. It is worth observing that the perception that something is wrong in the application behaviour is only based on the view of its interaction with the middleware. On the other hand, MIstRAL is unable to adapt the middleware due to neither changes in the environmental conditions nor the need of extending the middleware functionality. This is not possible because MIstRAL is not able to monitor the environmental conditions. It concentrates on monitoring behavioural aspects of the middleware execution by observing its execution trace.

MIstRAL works by reconfiguring the middleware by checking its execution trace against desired properties expressed in a temporal logic. The execution trace includes application's interactions with the middleware (not the application internal behaviour) and all actions performed by the middleware. For example, suppose a middleware starts to have a bad behaviour (i.e., a middleware component fails or does not respond to invocations), MIstRAL detects these problems and then triggers the needed reconfiguration actions at runtime. Another possibility is to identify that the application is not working properly based on the middleware execution trace.

Our unique contributions in this paper include: the moving of the adaptation process to an element outside the middleware; the adoption of a non-invasive approach in which existing middleware systems may be adapted without be modified; the use of a lightweight formal approach to guide the middleware adaptation; and the definition of formal properties that model anomalies in the functioning of object oriented middleware systems, which serve as basis for the adaptation process. Further minor contributions are related to the adoption of a pattern oriented approach to define the anomalies, and the actual implementation of an adaptive middleware in Java that uses the proposed approach.

This paper is organized as follows. Section II introduces basic concepts of middleware. Next, Section III presents the proposed approach to middleware reconfiguration. Section IV makes an evaluation of the proposed approach. Section V presents existing researches on design pattern formalisation. Finally, Section VI presents conclusions and some future work.

II. Adaptive Middleware

Middleware platforms facilitate the development of distributed applications by implementing distribution transparencies (e.g., access, location and failure) and providing distributed services (e.g., naming, security, and concurrency) to developers. In practice, the middleware hides from application developers the complexity of underlying communication and concurrency mechanisms, distribution and heterogeneity of operating systems, programming language, hardware and network.

Despite the great diversity of middleware models and products, they are usually structured into four layers [10]: *infrastructure* layer serves as a wrapper to low level communication and concurrency mechanisms of the operating systems; (ii) *distribution* layer is the heart of the middleware and defines the higher-level distributed programming models made available to application developers, e.g., it defines the programming abstractions (e.g., objects, components, procedures) used to build applications; *common services* layer that consists of generic services used by distribute application despite their domain (e.g., security, naming, transaction, concurrency); and specific services layer, which include domain-specific services such as telecommunication, e-commerce and e-learn.

Adaptive middleware is a particular kind of middleware whose behaviour can be modified at runtime motivated by the need of (i) adapting to changes of application's requirements, (ii) adapting to changes of environmental conditions, (iii) fixing middleware's bugs or (iv) extending the middleware functionality. Alterations in the application requirements are usually related to the need of a new/improved functionality provided by the middleware, e.g., an application that needs to be more secure demands a new stronger encryption algorithm implemented by the middleware security service. Changes in the environmental conditions can occur due to hardware failures and network traffic changes. Sometimes, however, the middleware adaptation can be motivated by the perception that something is wrong with the middleware behaviour. Finally, the need of adaptation is simply because the middleware should be extended due to implementation of a new feature of the middleware.

Adaptive middleware may be classified as static and dynamic [11]. Static middleware is one whose adaptation occurs at compile/start-up time, whilst in the dynamic middleware, the adaptation happens after the start-up time, i.e., at runtime. Adaptive middleware is typically implemented using four enabling software technologies: computational reflection, component-based design, aspect-oriented programming and software design patterns. By using computational reflection, the middleware can reason and alters its own behaviour. Meanwhile, in the component-based design, the middleware can be built through putting together software units that can be independently produced, deployed and composed. In aspect-oriented programming, the middleware can be composed of intervened cross-cutting concerns such as security, fault-tolerance, and so on. Finally, software design patterns specially defined to treat with adaptive concerns (known as adaptive design patterns) can be employed to develop the middleware, e.g., virtual component pattern, component configurator, invocation interceptor and chain of

interceptors.

III. MIstRAL

MIstRAL (MIddleware Reconfiguration Aid by formaLism) is an approach that uses formal elements, in a lightweight way, to guide the middleware reconfiguration process. This approach allows the development of adaptive middleware systems and has been defined using some basic principles:

- *Why*: MIstRAL allows the middleware reconfiguration motivated by the existence of middleware bugs and changes in the application's requirements and application's behaviour;
- *When*: the middleware reconfiguration should typically occur at runtime. However, MIstRAL may also be useful at middleware development time to detect undesired middleware behaviours;
- *How*: the reconfiguration process should be based on a formal tooling as the key enabling technology for developing adaptive middleware systems. Furthermore, the reconfiguration is compositional which means that four operations are used to reconfigure the middleware: add, remove, replace and reconnect structural middleware components as defined in [3];
- *Where*: MIstRAL is not part of the middleware, which means that the adaptation code is external to the middleware;
- *Middleware agnostic*: any middleware implementation using the idea of "chain of interceptors" (see Section II) should be reconfigurated using the proposed approach. It is worth observing that most existing middleware implementations uses chain of interceptors and allow their reconfiguration at runtime;
- *Solution itself is reconfigurable*: the solution itself should be reconfigurable, i.e., it is possible to change the temporal properties to be evaluated at runtime and the reconfiguration policies; and
- *Intensive use of design patterns*: intensive use of adaptive middleware design patterns that facilitates the reconfiguration process.

In practice, MIstRAL reconfigures the middleware by checking its execution trace against desired properties expressed in a temporal logic. For example, suppose a middleware starts to have a bad behaviour (i.e., a middleware component fails or does not respond to invocations), MIstRAL detects this problem and then triggers the needed reconfiguration actions.

Figure 1 presents a general overview of the proposed solution. The middleware execution trace is passed to MIstRAL that takes responsibility of checking behavioural properties and executing the needed reconfiguration without a complete stopping of the middleware. In the case the properties are not satisfied, the middleware is reconfigurated. The process is repeated according to some predefined reconfiguration policy, e.g., every 5 minutes, every 5 minutes in normal conditions and every minute right after the reconfiguration.

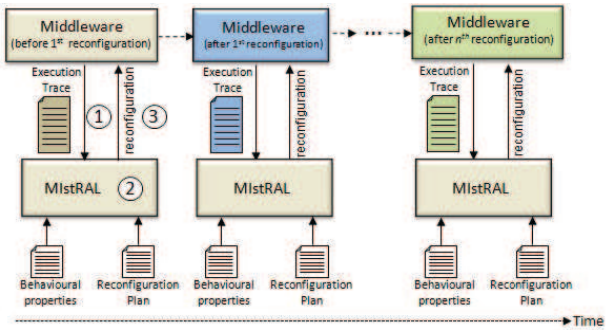

Figure 1. General Overview

The proposed approach involves three main steps: tracing, checking and reconfiguration. In the tracing step (1), an execution trace is generated by instrumenting the middleware to log its operations, e.g., the component responsible for marshalling messages must log proper information every time it is invoked, or any send/receive operation performed by the middleware must be registered. The checking step (2) occurs simultaneously to the middleware execution, in an independent way, and with the aid of a formal tooling. In this step, the generated trace is checked against middleware desired properties. Finally, the reconfiguration step (3) is executed by reconfiguring the middleware, e.g., by replacing a existing component or adding a new one.

More formally, given a middleware M, its execution trace T, and a set of desired middleware temporal properties P, T must satisfy P ($T \succ P$). If for some reason, T does not satisfy P ($T \not\succ P$), M is reconfigurated according to a reconfiguration plan R. R consists of a set of actions A performed on the middleware.

Next subsections present details of MIstRAL and formally introduce the aforementioned elements.

A. Basic Definitions

Prior to present the formalisation of elements involved in the MIstRAL approach, it is necessary to introduce some basic concepts including our notion of middleware, middleware component, component interface and invocation interceptor.

Definition 1 (Component Interface). An interface is the set of operations $I = \{op_1, op_2, ..., op_n\}$ performed by the middleware component that expresses its functionality.

It is worth observing that for our particular purpose, the signature of these operations, traditionally included in this kind of definition, is not considered here. We are only interested in the component behaviour, which is a sequence

of operations performed by the component whatever the input and output parameters, and their respective types.

Definition 2 (Middleware Component) A middleware component is a tuple $C =< I, H >$, where

- I is the component interface,
- H is the sequence of possible invocations to the component interface.

Definition 3 (Middleware) Middleware is a tuple $M =< S, D, F, B >$, where

- S is a set of components that make up the middleware and can not be reconfigurated at runtime,
- D is a set of components that make up the middleware and can be reconfigurated at runtime,
- $F \subseteq DXD$ is a relation that defines how the reconfigurable components are connected,
- B is the sequence of observable actions that can occur (be observed) at all components' interfaces of the middleware.

B. Tracing

Tracing is a key element of MIstRAL and refers to the way the trace is generated and its content. The execution trace is produced by registering observable actions executed by the middleware during its execution.

Definition 4 (Execution Trace) Middleware execution trace T is a sequence of actions $T \subseteq B \cup E$ performed by the middleware, where

- B is the sequence of observable actions that may occur at all components' interface of the middleware according to Definition 3, and
- E is the set of actions raised due to errors.

C. Checking

As mentioned before, the checking consists of verifying a temporal logic property on the middleware execution trace. The result of this verification defines whether the property is satisified ($T \succ P$) or not ($T \not\succ P$) by the middleware according to [12].

The properties adopted in this paper follows the set of property patterns defined in [13]: a given action does not occur within a scope (*absence*); a given action must occur within a scope (*existence*); a given action must occur *k* times within a scope (*bounded existence*); a given action occurs throughout a scope (*universality*); an action *a* must always be preceded by an action *b* (*precedence*); an action *a* must always be preceded by an action *b* (*response*); a sequence of actions $a_1, ..., a_n$ must always be preceded by a sequence of actions $b_1, ..., b_m$ (*chain precedence*); and a sequence of actions $a_1, ..., a_n$ must always be followed by a sequence of actions $b_1, ..., b_m$ (*chain of response*). By adopting these patterns and using the regular alternation-free μ-calculus [12], we consider the property patterns shown in Table I.

Table I
Property patterns

No	Description	Formula
P_1	"α_1 does not occur"	$[T*.\alpha_1]F$
P_2	"α_1 does not occur before α_2"	$[(\neg\alpha_2)^*.\alpha_1.(\neg\alpha_2)^*.\alpha_2]F$
P_3	"α_1 must occur before α_2"	$[(\neg\alpha_1)^*.\alpha_2]F$
P_4	"α_1 must occur after α_2	$[(\neg\alpha_1)^*.\alpha_2]\mu Y. < T > T[\neg\alpha_1]Y$
P_5	"α_1 must occur between α_2 and α_3	$[T^*.\alpha_2.(\neg(\alpha_1\alpha_3))^*.\alpha_3]F$
P_6	"α_1 must occur after α_2 until α_3	$[T^*.\alpha_2]\mu Y. < T > T[\alpha_3]F[\neg\alpha_1]Y$
$Deadlock$	No deadlock sequence is found	$[true*] < true > true$
$Livelock$	No livelock sequence is found	$[true*]muX.([\text{"}exit\text{"}]X)$

It is worth observing that given a middleware execution trace T, actions $\alpha_1, \alpha_2, \alpha_3 \in T$. Additionally, these properties must reflect the design pattern invocations as defined in [14].

D. Reconfiguration

As mentioned before, the reconfiguration is triggered when the verification of a temporal property (p_i) on trace produces a false result. As a consequence, a reconfiguration plan (r_i) must be executed to change the way the the dynamic components are connected (see Definition 3). Hence, we have

$$T \not\succ p_1 \rightarrow r_1$$
$$T \not\succ p_2 \rightarrow r_2$$
$$...$$
$$T \not\succ p_n \rightarrow r_n$$

In particular, four kinds of reconfiguration plans (r_i) can be performed: add a new component to configuration, to remove an existing component from configuration, to replace an existing component or to reconnect existing components of the configuration.

Definition 5 (Recofigurantion Plan) A reconfiguration plan is a function $r : FXF \mapsto F'$, where

- F, F' are relations that define how the middleware reconfigurable components are connected (see Definition 3).

Four reconfiguration plans are predefined as follows:

- $r_1(F, \{(c_i, c_n)\}) = F \cup \{(c_i, c_n)\}$, where $c_i \in D$ and c_n is the new component to be added to the middleware;
- $r_2(F, \{(c_i, c_o)\}) = F \setminus \{(c_i, c_o)\}$, where $c_i, c_o \in D$ and c_o is the component to be removed;
- $r_3(F, \{(c_i, c_o), (c_i, c_n)\}) = r_1(F, \{(c_i, c_n)\}) \circ r_2(F, \{(c_i, c_o)\})$, where $c_i, c_o \in D$, c_o is the component to be removed and c_n is the new component to be added;
- $r_4(F, \{(c_i, c_j), (c_k, c_l)\}) = r_1(F, \{(c_i, c_j)\}) \circ r_2(F, \{(c_k, c_l)\})$, where $c_i, c_j, c_k, c_l \in D$).

D is defined as shown in Definition 3.

E. Implementation

MIstRAL was implemented in Java and uses the CADP Toolbox[1] to check the execution trace. Figure 2 presents

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

a general overview of the proposed implementation. The adaptive middleware maintains a configuration file that defines the interceptors implemented by the middleware and whose access is mandatory in every invocation of a client/server application. It is worth observing that the list of interceptors and the configuration file correspond to the elements D (set of reconfigurable components) and F (how the reconfigurable elements are connected) as introduced in Definition 3.

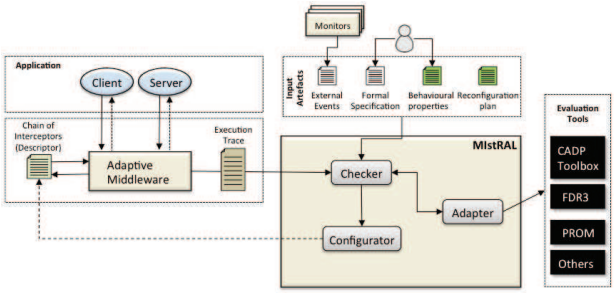

Figure 2. MIstRAL Architecture

The trace (see Section III-B) is obtained by instrumenting the middleware with a logger that takes responsibility of logging actions executed by the middleware (1). The trace is formatted by the *Processor* that formats the log according to the file format accepted by the CADP Toolbox. The *Processor* passes the formatted log to the *Checker* that invokes the CADP Toolbox to check the behaviour properties (2).

The response from CADP is evaluated by the *Checker*. If a reconfiguration is necessary ($T \not\succ P$), the *Checker* asks for the *Configurator* to perform the reconfiguration plan according to the property that was not satisfied (3). The reconfiguration plans (Definition 5) change the way the interceptors are connected and have been implemented as defined in the following Java interface definition:

```
public interface IReconfigurationPlans {
  public void addInt (int idx, MyInterceptor newInt);
  public void remInt(MyInterceptor oldInt);
  public void repInt(MyInterceptor oldInt, MyInterceptor newInt);
  public void recInt(MyInterceptor i1, MyInterceptor i2,
                     MyInterceptor i3, MyInterceptor i4);
}
```

The CADP was adopted as the formal toolbox, but other tools like PROM[2] or FDR3[3] should also be used. By using the CADP Toolbox, it is possible to check the following properties on the middleware trace (T): find deadlocks, find livelocks, find execution sequences, verify temporal formulas, reduce, compare, convert, find path to state, find non determinism, find unreacheable states, generate tests and evaluate performance. In particular, the reconfiguration as proposed in this paper is performed by verifying temporal formulas.

[2]http://www.processmining.org
[3]https://www.cs.ox.ac.uk/projects/fdr/

IV. Experimental Evaluation

In order to evaluate the proposed approach, we initially implemented an adaptive object-oriented middleware based on the remoting patterns [14] and a simple client/server application atop it (see Figure 3). The middleware was implemented using the following design patterns: *Requestor*, *Invoker*, *Client Proxy*, *Client Request Handler*(CRH), *Server Request Handler*(SRH), *Marshaller*, *ObjectID*, *Absolute Object Reference*, *Lookup*, *Invocation Interceptor* and *Invocation Context* [14]. Three different invocation interceptors compose the *Chain of Interceptors*(*Interceptor_1, Interceptor_2, Interceptor_3*) and they were implemented on the server side in such way that every remote invocation from the Client passes through them in an order defined in the *Descriptor)*. It is worth observing that according to Definition 3, the interceptors are the reconfigurable middleware components (D).

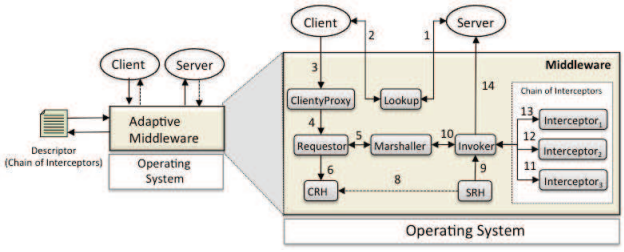

Figure 3. Adaptive object-oriented middleware

Figure 3 shows in a simplified way (for lack of space) the basic steps on how the middleware works. Steps 1 and 2 are the traditional ones in the client/server binding, whilst Steps 3, 4, 5, 6 and 7 are also very common in object-oriented middleware systems in which a client's invocation enters in the middleware through the *Client Proxy* and then is sent to the server side. On the server side, the invocation goes through Steps 8 and 9 before passes into the interceptor chain (Steps 10, 11 and 12). At this point, it is worth observing that the initial set of interceptors is defined according to the *Descriptor*. However, this descriptor can be reconfigurated at runtime according to the result of the execution trace analisys.

The experimental evaluation was divided into two steps: to show that the reconfiguration actually changes the middleware behaviour (Section IV-A); and to measure the overhead caused by the reconfiguration process, and to measure the time spent from the beginning of a bad behaviour and its proper fixing (Section IV-B).

A. Reconfiguration

To show that the reconfiguration actually works, we initially implemented three interceptors, where one of then (*Interceptor 2*) becomes slow and is replaced from time to time by another one (*Interceptor 3*) that is faster. Figure 4 shows the behaviour of the *response time* while the reconfigurations are triggered. The *response time* is the time lapsed between the client invocation to a remote object and the reception of the response. Every invocation passes through the chain of interceptors that is subject to the reconfigurations as mentioned before.

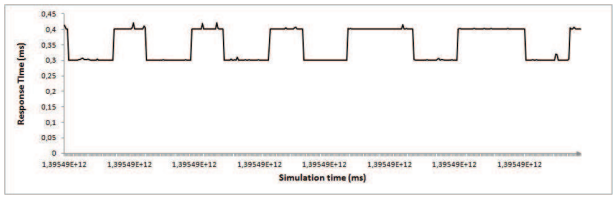

Figure 4. Reconfigurations of the chain of interceptors

As shown in Figure 4, the behaviour of the response time is very clear. When the slow interceptor is in the chain of interceptors, the response time is high. From the time this interceptor is replaced by another one (faster), the response time decreases until the next reconfiguration.

B. Performance Evaluation

As mentioned before, we also are interested in measuring two different performance aspects: the overhead caused by the reconfiguration process on the adaptive object-oriented middleware; and the time spent from the misbehaviour detection until the end of the reconfiguration that fixes this misbehaviour. These experiments were performed as shown in Figure 5 that presents the elements in which the invocation passes through.

We consider the performance metric *response time* (see Section IV-A) and the factor *reconfiguration mechanism* having two levels: *enabled* and *disabled*. The experiment executed in a Mac OS (OS X, version 10.8.5, processor 2.9 GHz Intel Core i7, 8 GB of RAM), where a client performs 10.000 invocations to a remote object, whose service time was set to 10 ms. The time shown in Figure 5 is the mean response time of the invocations.

As expected, this figure shows that there is an overhead in the response time when the reconfiguration mechanism is enabled. This overhead is 0,14 ms and it is only 1,45% of the service time. Hence, in practice, the overhead caused by the reconfiguration mechanism is very low when compared to the time spent with the business itself.

Second experiment focuses on the time spent with the reconfiguration. In practice, it is the time lasted from the detection that something is not working properly in the

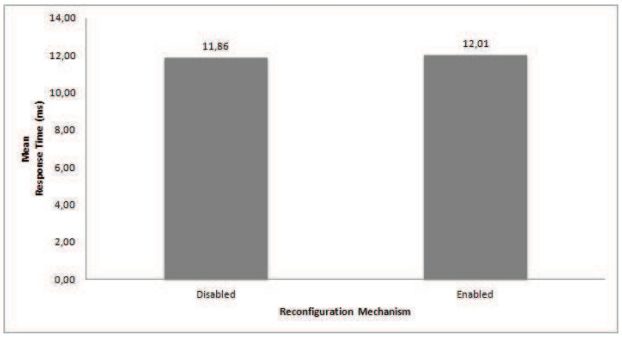

Figure 5. Overhead caused by the reconfiguration mechanism

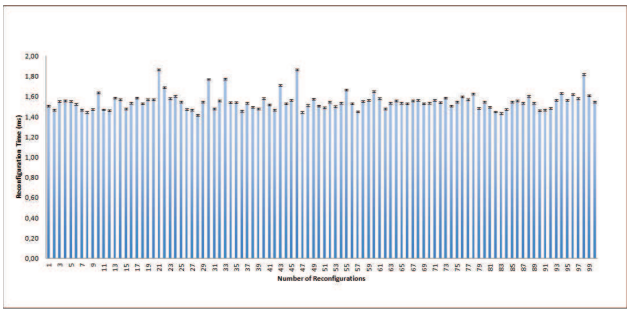

Figure 6. Reconfiguration time

middleware until the time the problem is fixed, i.e., the middleware is reconfigurated. Figure 6 depicts this time considering 100 consecutive reconfigurations. The meantime for reconfiguration is 1551,407 ms and standard deviation 27,302 ms, which shows that the reconfiguration time is very stable. The need of reconfiguration was artificially injected in the middleware by inserting an unexpected behaviour that was responsible for triggering the reconfiguration process. It is worth observing that the majority of this time is due to the time needed to the CADP Toolbox checks the desired property.

V. Related Work

Related works about what is being proposed in this paper may be organized into two main categories: works on the design and implementation of adaptive middleware systems and works on the use of formal description techniques (FDT) in non-adaptive middleware platforms.

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 [4], OpenORB [5] and OpenCom [6]. More recently, adaptive middleware have been built in several different application domains, such as large-scale power systems [7], onboard sattellite systems [8] and public transit system [9]. 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 [15]. Meanwhile, the use of formalisms in individual phases is widely found and it starts at the elicitation requirements phase [16]. Most common, however, is the adoption of FDTs associated to architectural aspects of the middleware [17] and at design phase [18]. Finally, there are some works on formalising the implementation phase [19] and when the middleware is already running [20].

Whatever the development phase, it is also possible to identify which aspects of middleware have been mainly formalised: middleware mechanisms and components [21], [22], middleware services [23], [24], [25] and middleware models [21], [26]. Orthogonal to services, mechanisms, components and models, the formalisation typically focuses on behavioural [27] and structural [28] aspects whatever is being specified. The behavioural aspects describe the temporal ordering of middleware actions, whilst the structural specifications describe the middleware elements and their relationships.

Finally, the formalisation has been done in three different ways: using existing general purpose formalism [29], designing and using a formalism specially planned for middleware [30] and through a formal framework/theory [31].

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.

VI. Conclusion and Future Work

This paper presented MIstRAL, an approach for building adaptive middleware systems based on the lightweight use of formal methods. MIstRAL use a formal approach as the enabling software mechanism to deal with how the reconfiguration is implemented. Meanwhile, MIstRAL does not insert the adaptation mechanism into the middleware. The mechanism is moved to an external component that decides and takes needed actions to reconfigure the middleware.

Our unique contributions in this paper include: the moving of the adaptation process to an element outside the middleware; the adoption of a non-invasive approach in which existing middleware systems may be adapted without be modified; the use of a lightweight formal approach to guide the middleware adaptation; and the definition of formal properties that model anomalies in the functioning of object oriented middleware systems, which serve as basis for the adaptation process. Further minor contributions are related to the adoption of a pattern oriented approach to define the anomalies, and the actual implementation of an adaptive middleware in Java that uses the proposed approach.

We are now using the same strategy with a publish/subscribe middleware, by extending the set of middleware properties and configuration plans and policies, which are being adapted to reflect the particularities of this middleware model such as delivery guarantee and extensive use of queues.

References

[1] A. Campbell, G. Coulson, and M. Kounavis, "Managing Complexity: Middleware Explained," *IT Professional*, vol. 1, no. 5, pp. 22–28, Sept.–Oct. 1999.

[2] D. Schmidt, D. Schmidt, and F. Buschmann, "Patterns, Frameworks, and Middleware: Their Synergistic Relationships," in *Proc. 25th International Conference on Software Engineering*, F. Buschmann, Ed., 2003, pp. 694–704.

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

[4] 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*, ser. Lecture Notes in Computer Science, J. Sventek and G. Coulson, Eds. Springer Berlin / Heidelberg, 2000, vol. 1795, pp. 121–143.

[5] 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.

[6] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware," in *Middleware 2001*, ser. Lecture Notes in Computer Science, R. Guerraoui, Ed. Springer Berlin / Heidelberg, 2001, vol. 2218, pp. 160–178.

[7] S. Rusitschka, C. Doblander, C. Goebel, and H.-A. Jacobsen, "Adaptive middleware for real-time prescriptive analytics in large scale power systems," in *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, ser. Middleware Industry '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:6.

[8] M. Fayyaz, T. Vladimirova, and J.-M. Caujolle, "Adaptive middleware design for satellite fault-tolerant distributed computing," in *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, June 2012, pp. 23–30.

[9] H. Rahnama, P. Kramaric, A. Sadeghian, and A. Shepard, "Self-adaptive Middleware for the Design of Context-aware Software Applications in Public Transit Systems," in *Proceedings of the 13th International Conference on Ubiquitous Computing*, ser. UbiComp '11. New York, NY, USA: ACM, 2011, pp. 491–492.

[10] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. John Wiley & Sons Ltd, 2006, vol. Volume 2.

[11] S. M. Sadjadi, "A Survey of Adaptive Middleware," Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, Tech. Rep., 2003.

[12] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free mu-calculus," *Science of Computer Programming*, vol. 46, no. 3, pp. 255 – 281, 2003, special issue on Formal Methods for Industrial Critical Systems.

[13] 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.

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

[15] M. Díaz, D. Garrido, L. Llopis, and J. M. Troya, "Designing distributed software with RT-CORBA and SDL," *Comput. Stand. Interfaces*, vol. 31, no. 6, pp. 1073–1091, Nov. 2009.

[16] M. Pradella, M. Rossi, D. Mandrioli, and A. Coen-Porisini, "A formal approach for designing CORBA based applications," in *Proceedings of the 22nd international conference on Software engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 188–197.

[17] M. Autili, C. Chilton, P. Inverardi, M. Kwiatkowska, and M. Tivoli, "Towards a connector algebra," in *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ser. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 278–292.

[18] N. S. Rosa, "Formalising middleware systems: A design pattern-based approach," in *37th IEEE Annual International Computer Software & Applications Conference*, 2013, pp. 658–667.

[19] N. S. Rosa, P. R. F. Cunha, and D. F. Sadok, "A methodology for realization of LOTOS specifications in the ANSAware," in *IFIP/IEEE International Conference on Distributed Platforms*, 1996, pp. 204–209.

[20] M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian, "Combining formal verification with observed system execution behavior to tune system parameters," in *Proceedings of the 5th international conference on Formal modeling and analysis of timed systems*, ser. FORMATS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 257–273.

[21] S. De, S. Chakraborty, D. Goswami, and S. Nandi, "Formalization of discovery and communication mechanisms of tuple space based mobile middleware for underlying unreliable infrastructure," in *2nd IEEE International Conference on Parallel Distributed and Grid Computing (PDGC)*, 2012, pp. 580–585.

[22] F. Arbab, "Puff, the magic protocol," in *Formal Modeling: Actors, Open Systems, Biological Systems*, ser. Lecture Notes in Computer Science, G. Agha, O. Danvy, and J. Meseguer, Eds. Springer Berlin Heidelberg, 2011, vol. 7000, pp. 169–206.

[23] D. Basin, F. Rittinger, and L. Viganò, "A formal data-model of the CORBA security service," in *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2001, pp. 303–304.

[24] B. Hafid, B. Mohamed, and E. hajji Said, "Verifying ODP trader function by using Event B," *International Journal of Computer Science*, vol. 7, no. 9, pp. 17–22, 2010.

[25] N. Venkatasubramanian, M. Deshpande, S. Mohapatra, S. Gutierrez-Nolasco, and J. Wickramasuriya, "Design and implementation of a composable reflective middleware framework," in *Proc. 21st Int Distributed Computing Systems Conf.*, 2001, pp. 644–653.

[26] A. Ressouche, J.-Y. Tigli, and O. Carrillo, "Toward validated composition in component-based adaptive middleware," in *Proceedings of the 10th international conference on Software composition*, ser. SC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 165–180.

[27] N. S. Rosa and P. R. F. Cunha, "A Formal Framework for Middleware Behavioural Specification," *Software Engineering Notes*, vol. 32, pp. 1–7, 2007.

[28] X. Renault, J. Hugues, and F. Kordon, "Formal modeling of a generic middleware to ensure invariant properties," in *Formal Methods for Open Object-Based Distributed Systems*, ser. Lecture Notes in Computer Science, G. Barthe and F. Boer, Eds. Springer Berlin Heidelberg, 2008, vol. 5051, pp. 185–200.

[29] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci, "The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems," in *Middleware 2011*, ser. Lecture Notes in Computer Science, F. Kon and A.-M. Kermarrec, Eds. Springer Berlin Heidelberg, 2011, vol. 7049, pp. 410–430.

[30] A. Ahern and N. Yoshida, "Formalising Java RMI with explicit code mobility," *Theoretical Computer Science*, vol. 389, no. 3, pp. 341 – 410, 2007, semantic and Logical Foundations of Global Computing.

[31] R. Baldoni, M. Contenti, S. Piergiovanni, and A. Virgillito, "Modeling publish/subscribe communication systems: towards a formal approach," in *Object-Oriented Real-Time Dependable Systems, 2003. (WORDS 2003). Proceedings of the Eighth International Workshop on*, 2003, pp. 304–311.