An Adaptive Middleware in Go

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

Gláucia Campos
Universidade Federal de Pernambuco
Centro de Informática
Recife, PE, Brazil
gmmc@cin.ufpe.br

David Cavalcanti
Universidade Federal de Pernambuco
Centro de Informática
Recife, PE, Brazil
djmc@cin.ufpe.br

ABSTRACT

Modern languages like Go has a set of innovative characteristics that create new possibilities for implementing adaptive middleware systems, such as runtime reflection, dynamic plugins, lightweight threads, and process algebra inspired channels. This paper explores those characteristics and combines them with software architecture concepts and lightweight formalisation to implement a framework for developing and executing adaptive middleware more safely. The framework is used to implement a middleware whose evaluation is performed in different scenarios to assess the impact of adaptation mechanisms on applications built atop it. In the end, it is possible to affirm that Go moves the development of adaptive middleware to a new baseline due to its simplicity, efficiency and reliability.

CCS CONCEPTS

• **Social and professional topics** → *Software selection and adaptation*; • **Theory of computation** → *Distributed computing models*; • **Software and its engineering** → *Software architectures*; *Formal methods*;

KEYWORDS

Adaptive Middleware; Go Language; Software Architecture; Formal Methods.

ACM Reference Format:
Nelson Rosa, Gláucia Campos, and David Cavalcanti. 2018. An Adaptive Middleware in Go. In *Proceedings of (ARM'18).* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3289175.3289176

1 INTRODUCTION

Whatever the application domain, the implementation of an adaptive middleware is impacted by the primary issues commonly found in the design of adaptive software systems [1]: *Why* do we have to adapt? *When* to adapt? *Where* is the need of change? *What* artefacts need to be modified? *How* is the adaptation performed?. As identified in [2], additional concerns have their origin in the middleware domain itself. Firstly, the adaptation can be motivated by changes in both the application's requirements (above the middleware) and infrastructure conditions (below the middleware). Secondly, the right time to trigger the middleware adaptation should consider the application's state. Finally, the middleware adaptation is also more critical in the sense that changes can affect both the middleware itself and the application built atop of it.

The efforts to design and implement adaptive middleware are not recent [3][4]. However, the emergence of new application domains (e.g., Internet of Things [5] and cloud computing [6]), new technologies like process mining [2], improvements on model checkers performance [7] and increasing adoption of autonomic computing concepts have been responsible for renewing the challenges and possibilities of developing safer and more efficient adaptive middleware systems.

The emergence of Go [8] motivated this initial experience to explore its powerful standard library, concurrency model (goroutines and channels) and dynamic facilities (plugins and runtime reflection) to face the challenges mentioned early. Instead of starting from the scratch, however, the use of Go comes along with a new implementation of our previous Java adaptive middleware framework [2] (*MidArch*). The Go implementation, namely *gMidArch*, is also centred on the use of software architecture principles and lightweight formalisation. Meantime, the new implementation extends the formalisation capabilities through the support of a new formal operator (choice), an adaptation mechanism faster and more flexible, and changes in the core communication mechanism used by architectural elements.

The rise of modern languages like Go helps to boost the implementation of adaptive middleware. In practice, the potential of Go is a two-way street between the implementation and design. Being a programming language, Go enables us to advance on implementation issues while some facilities already provided by the language allows improvements on the middleware design. For example, the concurrency model that includes goroutines (lightweight threads) and channels inspired on Communicating Sequential Processes (CSP) [9] enormously facilitate the implementation of concurrency issues. Meanwhile, the existence of dynamic plugins simplifies the design of the adaptation mechanisms.

The evaluation of *gMidArch* focused on comparing the performance of an application that uses both a middleware instance implemented using *gMidArch* and the Go package RPC. To better assess the impact of the adaptation mechanism over the application, we also compared the performance of the middleware instance when the adaptation mechanism is turned on/off.

Go is a widely adopted programming language whose potentials and drawbacks have been commonly discussed in a great variety of forums and existing literature. Instead of considering pros and cons of the language or trying to compare its mechanisms with the Java ones, this paper makes an effort to bring to light some benefits

ARM'18, December 2018, Rennes, France

ACM ISBN 978-1-4503-6132-3/18/12…$15.00
https://doi.org/10.1145/3289175.3289176

of using Go to face *how* to implement the adaptation mechanisms of a middleware.

The rest of this paper is organised into five sections. Section 2 briefly describes *MidArch*. Next, Section 3 presents details of *gMidArch*. Next, Section 4 makes a performance evaluation of *gMidArch*. Section 5 presents existing researches on adaptive middleware. Finally, Section 6 presents conclusions and some future work.

2 MIDARCH

MidArch (adaptive *Mid*dleware aid by software *Arch*itecture) [7] is a solution to help middleware developers to implement and execute adaptive middleware in a safe way. It uses lightweight formalisation, includes a software architecture-based framework and an execution environment. The framework facilitates the *middleware development* and the lightweight adoption of formal methods (CSP) integrated into the framework enables us to verify behavioural properties before and while the middleware executes. At runtime, the *middleware execution* is managed by an environment that implements a MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) feedback loop [10] and uses process mining tools [11] to decide the right time to adapt.

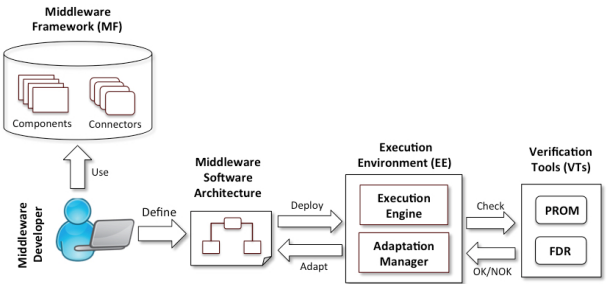

Figure 1: General overview of *MidArch*

As shown in Figure 1, using the middleware framework, developers need to define a software architecture in a Java-based Architecture Description Language (ADL) to implement the adaptive middleware. The framework consists of a set of components and connectors implemented in Java, annotated with CSP specifications and freely reused and composed in a software architecture according to the needs of the middleware developer. Having defined the architecture, it is then verified to check behavioural (using FDR[1]) and structural properties, deployed and executed by the execution environment. At runtime, middleware execution logs are continuously verified by process mining tools (PROM[2]) to trigger adaptations when a middleware property is violated. In this case, an adaptation plan is generated and executed to change the middleware architecture that is verified again and redeployed.

3 GMIDARCH

As mentioned before, *gMidArch* was developed around software architecture concepts and lightweight formalisation. The framework explores the Go characteristics with two main purposes: to extend *MidArch* conceptually to make viable the use of more complex architectural components and connectors (extensions); and to implement *MidArch* concepts in a more cleaner, flexible and efficient way (improvements).

In practice, all elements shown in Figure 1 were reimplemented in Go[3]. The extensions are related to the Middleware Framework as components and connectors have their CSP behaviours specified with a richer CSP subset; the Adaptation Manager was redesigned to include three adaptation strategies; and a new repository was included with the component/connector plugins used at runtime. The improvements focused on the Execution Environment whose implementation makes extensive use of goroutines and channels to improve the performance and simplify its design.

These extensions and improvements are presented in details in the next subsections.

3.1 External choice operator

The lightweight formalisation is one of the essential characteristics of *MidArch*. The formalisation consists of defining behavioural models in CSP that describe how components/connectors belonging to the framework behave while executes. This behaviour specification is used to generate a state machine (graph) executed by the Execution Engine (shown in Figure 1).

In *MidArch*, the specification of behaviours can only use the prefix operator ($\rightarrow$), e.g., $e \rightarrow P$ informally means that the process performs the event e and then P runs. This operator enables us to specify behaviours as a sequence of actions where one is executed after another. For example, the behaviour of a marshaller can be easily specified like $B = invP.e1 \rightarrow i\_PosInvP \rightarrow terP.e1 \rightarrow B$. In this behaviour expression, the marshaller receives an *invocation* from external component $e1$ (*invP.e1*), (un) marshalls it (*i_PosInvP*), *terminates* the processing (*terP.e1*) by sending a response to $e1$, and behaves like B again.

While useful, the prefix operator limits the kind of behaviours that can be specified. In the previous example, the marshaller cannot receive a request from other components while waits for a request from $e1$ and it is not possible to specify the situations in which the component fails. In practice, these specifications should typically include alternatives or choices to express more complex behaviours.

Having this limitation in mind, the use of Go created the opportunity to enrich the subset of CSP used to specify behaviours with the inclusion of the external choice operator ([]). In practice, the existence of the Go *select* statement made possible to implement a more sophisticated Execution Engine that is now capable of executing behavioural specifications including the choice operator.

The choice informally works as follows: $a \rightarrow B1[]b \rightarrow B2$ offers the choice of a and b, i.e., in a given time, event a or b can be chosen. For example, using this operator, the marshaller behaviour can now be specified as follows:

$$
\begin{aligned}
B \;=\; & invP.e1 \rightarrow i\_PosInvP \rightarrow terP.e1 \rightarrow B \\
& [] \\
& invP.e2 \rightarrow i\_PosInvP \rightarrow terP.e2 \rightarrow B \\
& [] \\
& i\_failure \rightarrow i\_fixProblem \rightarrow B
\end{aligned}
$$

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

[3] https://github.com/nsrosa70/midarch-go.git

As shown in a straightforward way in Figure 2, the semantics of the choice operator leads to the generation of more complex state machines, i.e., state machines whose nodes can have two or more edges.

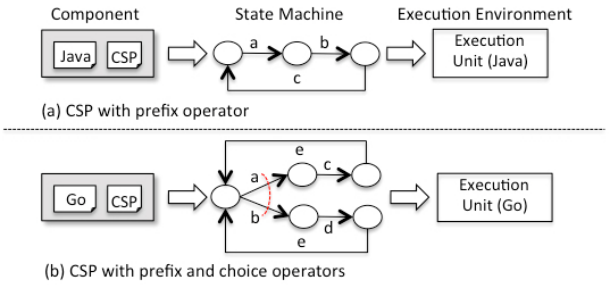

Figure 2: Practical view of the CSP extension

While the generation of the state machines shown in Figures 2(a) and 2(b) is not a problem in both Java and Go, Java does not support an execution engine able to execute the second machine. The second machine (2(b)), however, requires the selection of an event (e.g., *a* and *b*) from a list of several events (adjacent edges) enabled simultaneously to be executed.

The Go *select* statement lets a goroutine to wait on multiple communication operations (events). The statement blocks until one of its cases can execute, then it runs that case. It chooses randomly if multiple cases are ready. The *select* statement made viable the implementation of an execution engine capable of executing the second state machine. The use of reflection was also essential because the engine does not know the branches (*a* and *b*) statically and needs to define them at runtime. The list of edges (events) is created while the unit is traversing the state machine.

In *gMidArch*, Algorithm 1 is implemented by the execution units managed by the execution engine.

Algorithm 1 Execution of the State Machine

```
1:  function ExecuteStateMachine(graph)
2:      node = 0
3:      while true do
4:          edges = AdjacentEdges(graph,node)
5:          if len(edges) = 1 then                    ▷ Prefix operator
6:              action = Event(edges[0])
7:              node = NextNode(edges[0])
8:          else                                      ▷ Choice operator
9:              chosen = Choice(edges)                ▷ Function Choice
10:             action = Event(edges[chosen])
11:             node = NextNode(edges[chosen])
12:         end if
13:         Execute(action)
14:     end while
15: end function
```

This algorithm essentially traverses the graph whose nodes are the states of a component/connector, and the edges are labelled with actions executed by it. Given a node, it is necessary to get the node's adjacent edges. If there is a single adjacent node (Line 5), it means that the next action to be executed is the one associated to the edge (Line 6). Otherwise, there is a branch in the graph that indicates the use of the choice operator (Line 8). In this case, function *Choice* (Line 9) must select an edge from a list (*edges*). The next node depends on the selected edge (Line 11). Finally, in both cases, the action associated with the edge is executed (Line 13).

The implementation of function *Choice* in Go is shown in the following code snippet:

```
1:  func Choice(chosen *int, edges []wgraph.Edge) {
2:      cases := make([]reflect.SelectCase, len(edges))
3:      var value reflect.Value
4:      for i := 0; i < len(edges); i++ {
5:          cases[i] = reflect.SelectCase{Dir:
6:                      reflect.SelectRecv, Chan:
7:                      reflect.ValueOf(*edges[i].Action.P2)}
8:      }
9:      *chosen, _, _ := reflect.Select(cases)
10: }
```

The input parameters are a pointer to be set when a case is selected and the set of selectable edges, e.g., events *a* and *b* shown in Figure 2. In Lines 2-8, this implementation uses reflection to create a set of case clauses (events) and the *select* statement that blocks until one of the events occurs (Line 9).

3.2 Runtime replacement

gMidArch implements three different kinds of adaptation strategies, namely corrective, proactive and evolutive. A corrective adaptation is triggered when a problem in the middleware execution is detected and needs to be fixed, e.g., an error to establish a connection. Currently, we use process mining techniques to trigger this kind of adaptation. The second kind of adaptation (proactive) is triggered through the use of probabilistic models, e.g., if a middleware component is close to reaching a performance saturation, the adaptation is triggered. Finally, evolutive adaptations occur when a new version of an existing component/connector becomes available. Whatever the adaptation strategy, all of them typically demand the replacement of a running element by another one. It is worth observing that after deploying the software architecture into the execution environment, any component/connector can be potentially replaced.

The replacement process mainly consists of loading the new component, stopping the old one and starting the new one. Using CSP annotations associated to components and connectors, two additional steps are performed: the checking of behavioural compatibility between the old and new component and the verification if the new component injects an undesirable behaviour in the architecture.

In *gMidArch*, all architectural elements (component and connectors) have a basic implementation (loaded statically when the architecture is deployed in the execution environment) and a plugin version (loaded dynamically). Go plugins allow developers to implement loosely coupled programs whose components can be dynamically loaded and bound at runtime.

Figure 3 shows a general overview of the replacement process. After deployed (1), a software architecture can be adapted anytime

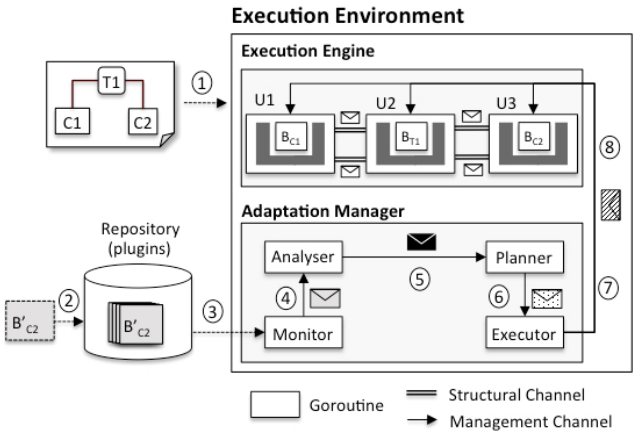

Figure 3: Goroutines, channels and plugins

as mentioned before. The deployment means to create and start an execution unit for each architectural element (*U1,U2 and U3*). The unit works as a kind of container where the component/connector executes. The execution unit traverses the state machine and invokes the events defined in the element's behaviour (B_{C1}, B_{T1} and B_{c2}) as defined in Algorithm 1. In parallel to the deployment and execution of the software architecture, the *Adaptation Manager* starts the MAPE-K components, namely Monitor, Analyser, Planner and Executor. Then, as soon as a new plugin is available (e.g., a plugin for component B_{C2} whose behaviour is B'_{C2}), the monitor detects its presence (3) and then passes this information to the analyser (4). It is worth observing that the plugin also has a CSP behaviour (B'_{C2}) in a similar way to the non-plugin component. Next, the analyser checks the compatibility of the current behaviour and the behaviour of the new plugin. If they are compatible, the analyser generates a new CSP specification, including the behaviour of the new plugin, which specifies the new behaviour of whole architecture. If the specification has not an undesired behaviour like deadlock, it means that the new plugin can be deployed. Next, the analyser informs the planner about the new plugin (5). The planner creates an adaptation plan that needs to be executed to carry out the adaptation. The plan is passed (6) to the executor. The executor notifies the execution unit (e.g., *U3*) that stops, loads the new behaviour (B'_{C2}) and discards the old one (B_{C2}).

As mentioned before, the replacement process may occur continuously. However, some points must be observed that are not apparent in Figure 3. Firstly, the detection of a new plugin is simple, as they are stored in a repository. From time to time (configurable), the monitor checks the repository for new plugins. Secondly, two components are compatible if they have the same type (Go checking) and if the behaviour of the new plugin refines (in a particular semantic model) the behaviour of the old element (CSP checking). Using FDR, this checking is performed through an assertion like *assert* $B1$ $[T = B2$, where $B1$ and $B2$ are the behaviours of the new and old elements, respectively. Thirdly, the behaviour replacement only occurs when the old element is in its initial state that works as a quiescent state, i.e., no pending requests. Fourthly, only stateless elements can be replaced in the current version. Fifthly, as will be explained in Section 3.3, each execution unit and MAPE-K element is defined as a goroutine (lightweight threads) that communicates through synchronous channels. It is worth observing that the channels between units are defined dynamically according to the middleware software architecture, e.g., if two components are connected in the architecture, channels are created between them. Meanwhile, the communication channels between MAPE-K components are statically defined in the execution environment. Sixthly, solely the unit whose behaviour is replaced needs to be stopped. Finally, it is worth observing that the Go plugins implement two additional functions in relation to its non-plugin version: *GetTypeElement()* and *GetBehaviourExp()*. They return the plugin type and behaviour expression (CSP) of the plugin, respectively. Those functions are necessary for the type and behavioural compatibility checks mentioned before.

3.3 Implementation improvements

In addition to the extensions presented in the previous section, some improvements in the implementation itself became possible and are related to the extensive use of goroutines and channels.

As said before, execution units (see Figure 3) and MAPE-K elements were implemented as goroutines, while they were Java threads in *MidArch* version. Goroutines have some characteristics like faster startup time than Java threads and a smaller number of operating system threads as there is not a 1:1 mapping between them. Goroutines come with built-in primitives to safe communication between them, the so-called channels.

Due to the diversity of goroutines used, it was necessary to define a set of different types of channels to accommodate the ways execution units and MAPE-K elements communicate. As depicted in Figure 3, units have two different kinds of channels, namely *management* and *structural* channels. Management channels are used by the execution engine to send management actions to units like stop and resume, e.g., when a component/connector needs to be replaced by a new plugin, the unit receives a stop signal, the new behaviour and the resume command. Meanwhile, structural channels are used in the communication with other units to reflect the software architecture attachments.

Figure 3 also shows the channels used for executing a software architecture. This figure has channels that transport different kinds of information: unit-to-unit channel transports messages exchanged by architectural elements; monitor-to-analyser transports log information and list of plugins; analyser-to-planner transports the result of the analysis; planner-to-executor transports adaptation plans; and executor-to-unit channels transport management commands. As shown in this figure, each execution unit ($U1$, $U2$ and $U3$) has 2-4 structural channels and one management channel to communicate with the executor.

A key aspect to understand how the software architecture is executed in practice is to observe the relationships between goroutines, channels, CSP specifications and state machines (graphs). As mentioned in Section 3.1, the CSP specification that describes the behaviour of each architectural element (set of temporally ordered actions) is translated into a state machine whose edges are labelled by actions executed by the element.

There are two kinds of actions, namely internal and external actions. Internal actions are those executed by the element and whose computation does not involve interaction with other elements. This kind of action refers to the "business" of the element, e.g., the marshaller has two internal actions to marshall and unmarshall messages. Meanwhile, external actions are those executed by the element that involves the interaction with other elements. For example, the marshaller has two external actions: one to receive a request from an outer element and another to send the result of the marshalling.

There is a 1:1 mapping between external actions performed by the architectural element and the unit's structural channels. Hence, when the unit transverses the graph, each action within the edge is associated with a function whose execution is to send/receive something via the structural channel. Four different external actions are allowed in *gMidArch*: *invR* and *terR* to send a request and receive a response from an external element, respectively; and *invP* and *terP* to receive a request and send a response to another element, respectively.

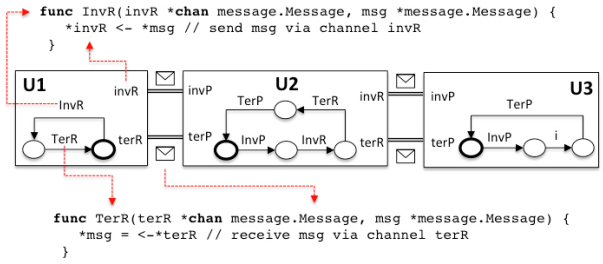

Figure 4: Go channels

In Figure 4, as execution unit *U1* transverses the state machine, the actions within the edges are executed. The execution of the first action (*InvR*) means to invoke function *InvR* that places a message (*msg*) into channel *invR*. Channels *invR* and *invP* are shared between *U1* and *U2* and when *U1* executes *InvR* and *U2* executes *InvP*, Go routines *U1* and *U2* synchronises and the message moves from *U1* to *U2*. The same process repeats in relation to other units and actions.

4 EVALUATION

The objective of the performance evaluation was to compare the performance of a simple client-server application built atop three different middleware flavours: the native Go package RPC (RPC), a non-adaptive instance of *gMidArch* (Non-Adaptive *gMidArch*), and an adaptive configuration of *gMidArch* (Adaptive *gMidArch*). Adaptive *gMidArch* uses the evolutive adaptation strategy in which an adaptation (replacement) occurs when a new plugin of component *Invoker* becomes available.

The metric used in the experiments was the *response time*, which is measured on the client side and refers to the time elapsed between the client makes a request to a remove function and receives a response from the server. As the focus is on the middleware, the remote function is one that recursively calculates a Fibonacci sequence number (n=38), which is a highly intensive processing task. All elements of application and middleware were deployed in three official Docker containers for Go[4] that run a client, a server and the *gMidArch* naming service. The containers executed on a MacBook Pro with a 2,9 GHz Intel Core i7, 8 GB RAM, and MacOS Sierra, version 10.12.6.

In the experiments, we varied the number of requests performed by the client: 1.000, 5.000 and 10.000. We also configurated the versioning interval to variate the frequency in which a new plugin version becomes available, e.g., 1 second, 30 seconds and 300 seconds.

Figure 5 shows the results[5] of the experiment in which the versioning interval was set to 1 second (very often adaptions), the number of requests was 10.000, and we varied the middleware flavour. This boxplot shows that the mean response times (dashed lines) of RPC, Non-adaptive and Adaptive *gMidArch* are 319,27 ms, 293,12 ms and 293,98 ms, respectively. A T-Test was also used to compare all paired samples and showed that the Non-Adaptive *gMidArch* is faster than both RPC and Adaptive *gMidArch*. Furthermore, the mean response time of the adaptive *gMidArch* is 0,29% higher than the Non-adaptive version. Those numbers show that a middleware built using *gMidArch* is very efficient and the impact of the adaptation mechanisms, even in a scenario with a high number of adaptations (one every second), is very low.

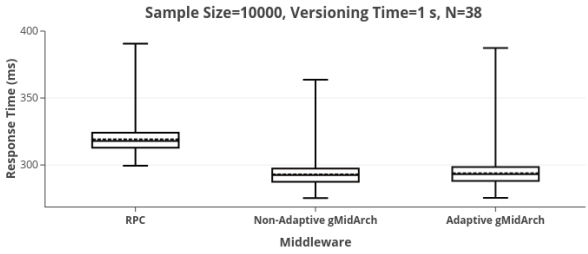

Figure 5: Comparative performance

5 RELATED WORK

Solutions to facilitate the implementation of middleware systems are also not recent. Middleware frameworks such as Quarterware [12] and PolyORB [13] were pioneers in this topic. Even sharing the idea of an implementation framework, these solutions have neither formal elements nor focus on adaptive middleware.

Following the same idea of developing middleware by combining existing middleware elements, Issarny [14] and Costa [15] focus on composing service specifications to define software architectures and combining building blocks (metamodels) to create middleware configurations, respectively. Unlike *gMidArch*, they do not address adaptive issues, nor use formal techniques nor cover implementation and execution activities.

Moving to the integration of formal verification as proposed by *gMidArch*, Caporuscio [16] defined a methodology to simplify the verification of behavioural properties of middleware-based software architectures. However, the formalisation is restricted to the software architecture, and it neither covers adaptation issues nor runtime.

[4]https://hub.docker.com/_/golang/
[5]Experimental data and graphs available at https://plot.ly/~nsrosa/21/

Apart from the middleware community, Rainbow [17] is a framework that provides supporting mechanisms for self-adaptation and a language (Stitch) that can be used to codify adaptation techniques. It implements a MAPE-K feedback loop to manage and trigger the adaptation when a structural property is not satisfied. *gMidArch* also checks behavioural and quality properties in addition to structural ones. Furthermore, *gMidArch* is a middleware-specific framework and uses lightweight formalisation.

The combination of formal methods and the adaptation mechanism has a variety of faces: incorporation of formal methods into a reflective component model to verify whether a runtime adaptation would violate structural constraints specified in ALLOY [18]; use of Petri nets to formally check if a given configuration complies with functional and non-functional properties of the system [19]; and use of an automata-based language in the feedback loop of the adaptive middleware [20]. These approaches use formal methods, but they have not focused on taking benefits of formalisms while the middleware is being developed.

6 CONCLUSION AND FUTURE WORK

Our unique contributions in this paper include the full re-implementation of *MidArch*in Go, the inclusion of a new CSP operator (external choice) to specify the behaviour of architectural elements and improvements of the adaptation mechanism. The extensive use of goroutines, channels and plugins also led to a very efficient implementation of the middleware whose performance is better than the Go package RPC in the evaluated scenarios.

Due to facilities provided by the Go concurrency mechanism, a critical next step is to extend the CSP annotations with parallel operators that allow the formalisation of composite components/connectors. While developed for implementing RPC-based middleware, it is also necessary to extend the proposed framework for building publish/subscribe middleware systems and define a pure ADL (Architecture Description Language). Finally, we are starting to work on a more elaborated/realistic evaluation of the solution to better identify their strengths and weaknesses, e.g., how the verification scales with the increase of the middleware complexity, how it reduces the programming effort.

REFERENCES

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1516533.1516538

[2] N. Rosa, "Middleware adaptation through process mining," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, March 2017, pp. 244–251.

[3] 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, 2000, vol. 1795, pp. 121–143.

[4] 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.

[5] I. Sarray, A. Ressouche, D. Gaffé, J.-Y. Tigli, and S. Lavirotte, "Safe Composition in Middleware for the Internet of Things," in *Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT*, ser. M4IoT 2015, 2015, pp. 7–12.

[6] A. Rafique, D. Van Landuyt, V. Reniers, and W. Joosen, "Towards an Adaptive Middleware for Efficient Multi-Cloud Data Storage," in *Proceedings of the 4th Workshop on CrossCloud Infrastructures & Platforms*, ser. Crosscloud'17, 2017, pp. 4:1–4:6.

[7] N. Rosa, G. Campos, and D. Cavalcanti, "Using software architecture principles and lightweight formalisation to build adaptive middleware," in *Proceedings of the ARM 2017*, ser. ARM '17, 2017, pp. 1:1–1:7.

[8] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, 1st ed. Addison-Wesley Professional, 2015.

[9] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[10] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., Jun. 2005.

[11] W. van der Aalst, *Process Mining - Data Science in Action*. Springer, 2016.

[12] A. Singhai, A. Singhai, A. Sane, and R. Campbell, "Quarterware for Middleware," in *Proc. 18th International Conference on Distributed Computing Systems*, A. Sane, Ed., 1998, pp. 192–201.

[13] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon, *PolyORB: A Schizophrenic Middleware to Build Versatile Reliable Distributed Applications*. Springer, 2004, pp. 106–119.

[14] V. Issarny, C. Kloukinas, and A. Zarras, "Systematic aid for developing middleware architectures," *Commun. ACM*, vol. 45, no. 6, pp. 53–58, Jun. 2002.

[15] F. M. Costa, K. A. Morris, F. Kon, and P. J. Clarke, "Model-Driven Domain-Specific Middleware," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 1961–1971.

[16] M. Caporuscio, P. Inverardi, and P. Pelliccione, "Compositional verification of middleware-based software architecture descriptions," in *Proceedings. 26th International Conference on Software Engineering*, May 2004, pp. 221–230.

[17] D. Garlan, B. Schmerl, and S.-W. Cheng, *Software Architecture-Based Self-Adaptation*. Springer, 2009, pp. 31–55.

[18] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe, "An automated formal approach to managing dynamic reconfiguration," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, Sept 2006, pp. 37–46.

[19] M. García–Valls, D. Perez–Palacin, and R. Mirandola, "Extending the verification capabilities of middleware for reliable distributed self-adaptive systems," in *INDIN 2014*, July 2014, pp. 164–169.

[20] A. N. Sylla, M. Louvel, and E. Rutten, "Combining transactional and behavioural reliability in adaptive middleware," in *ARM 2016*. New York, NY, USA: ACM, 2016, pp. 5:1–5:6.