AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications

Mila Dalla Preda[1], Saverio Giallorenzo[2], Ivan Lanese[2], Jacopo Mauro[2], and Maurizio Gabbrielli[2]

[1] Department of Computer Science - Univ. of Verona
[2] Department of Computer Science and Engineering - Univ. of Bologna / INRIA

Abstract. We present AIOCJ, a framework for programming distributed adaptive applications. Applications are programmed using AIOC, a choreographic language suited for expressing patterns of interaction from a global point of view. AIOC allows the programmer to specify which parts of the application can be adapted. Adaptation takes place at runtime by means of rules, which can change during the execution to tackle possibly unforeseen adaptation needs. AIOCJ relies on a solid theory that ensures applications to be deadlock-free by construction also after adaptation. We describe the architecture of AIOCJ, the design of the AIOC language, and an empirical validation of the framework.

1 Introduction

Adaptation is a main feature of current distributed applications, that should live for a long time in a continuously changing environment. Anticipating all the possible adaptation needs when designing an application is very difficult, thus the approaches able to cope with *unforeseen* adaptation needs are the most interesting. Also, for distributed applications like the ones that we consider, it is important to ensure deadlock-freedom (according to [1] about one third of concurrency bugs in real applications are deadlocks). While many techniques ensuring deadlock freedom exist in the literature, e.g., [2–4], to the best of our knowledge, none of them deals with adaptive applications. Indeed, most of the approaches to adaptation offer no guarantee on the behaviour of the application after adaptation [5–7], or they assume to know all the possible adaptations in advance [8], thus failing to cope with unforeseen adaptation needs.

Here we present AIOCJ, a prototype implementation of a framework for programming adaptive distributed applications that guarantees deadlock-freedom by construction (the theoretical foundations ensuring this property are discussed in [9]). AIOCJ is composed of two parts: (*i*) a domain-specific language, called Adaptive Interaction-Oriented Choreographies (*AIOC*) and (*ii*) an *adaptation middleware* that supports adaptation of AIOC programs.

The AIOC language describes applications from a global point of view following the *choreography* paradigm. This paradigm has been applied in different contexts, see, e.g., [2, 10–13], but we are not aware of other tools based on it and targeting adaptive applications. A choreography defines the interactions among the processes of a distributed application. AIOC main innovation consists in two constructs supporting

B. Combemale et al. (Eds.): SLE 2014, LNCS 8706, pp. 161–170, 2014.

adaptation: *scopes* and *adaptation rules*. A scope delimits code that may be adapted in the future. An adaptation rule provides new code to replace the one in a given scope. Interestingly, in AIOCJ, adaptation rules can be defined and inserted in the framework while the application is running, to cope with adaptation needs which were not foreseen when the application was designed or even started.

The code below shows a toy AIOC program (left) and an adaptation rule applicable to it (right). On the left, Lines 2-4 define a scope in which the local variable `msg` of process `user` is set to `"Hello World"`. The keyword `prop` defines properties of the scope (prefixed by `N`). In this case, the `name` property is set to `"hello_world"`. At Line 5 `user` sends the content of `msg` to a second process (`display`), that stores it in its local variable `msg`. On the right, Lines 2-3 define the *applicability condition* of the rule, i.e., the `name` property of the scope should be set to `"hello_world"`, and the environmental property `E.lang` should be equal to `"it"`. Line 4 shows the code that will replace the one of the scope, i.e., variable `msg` of `user` will be set to `"Ciao Mondo"` (Italian for "Hello World").

```
1  aioc {
2   scope @user{
3    msg@user = "Hello World"
4   } prop { N.name = "hello_world"};
5   send: user( msg ) -> display( msg ) }
```

```
1  rule {
2   on { N.name == "hello_world"
3      and E.lang == "it" }
4   do { msg@user = "Ciao Mondo" }
5  }
```

An AIOC program describes a full distributed application. AIOCJ generates, for each distributed process, a service written in Jolie [14, 15], a Service-Oriented orchestration language.

The adaptation middleware consists of a set of adaptation rules stored in multiple, possibly distributed, *adaptation servers*, and of an *adaptation manager* that mediates the interactions between the adaptive application and the various adaptation servers.

Structure of the paper. Section 2 presents an overview of the AIOCJ framework, while Section 3 describes its implementation. Section 4 shows a preliminary validation of the framework, with tests on the performances of AIOCJ. In Section 5 we discuss related work and future directions of research. A short demo of the use of the framework is available in the companion technical report [16].

2 Overview: The AIOCJ Framework

This section first defines the architectural model that supports adaptation of AIOCJ applications and then it introduces the syntax of the AIOC language via an example (for a formal presentation of AIOC syntax and semantics see [9]).

The AIOCJ Middleware. We consider applications composed of processes deployed as services on different localities, including local state and computational resources. Each process has a specific duty in the choreography and follows a given protocol. Processes interact via synchronous message passing over channels, also called *operations*. Adaptation is performed by an adaptation middleware including an adaptation manager and some, possibly distributed, adaptation servers. The latter are services that act as repositories of adaptation rules and may be (manually) added or removed at runtime. Running adaptation servers register themselves on the adaptation manager. The running application may interact with the adaptation manager to look for applicable adaptation rules. The effect of an adaptation rule is to replace a scope with new code that answers a given adaptation need. The adaptation manager checks the rules according to the registration order of the adaptation servers, returning the first applicable rule, if any.

The AIOC Language. The language relies on a set of roles that identify the processes in the choreography. Let us introduce the syntax of the language using an example where Bob invites Alice to see a film.

```
1  include isFreeDay from "calendar.org:80" with http
2  include getTicket from "cinema.org:8000" with soap
3  preamble {
4    starter: bob
5    location@bob = "socket://localhost:8000"
6    location@alice = "socket://alice.com:8000"
7    location@cinema = "socket://cinema.org:8001" }
8  aioc{
9    end@bob = false;
10   while( ! end )@bob{
11     scope @bob {
12       free_day@bob = getInput( "Insert your free day" );
13       proposal: bob( free_day ) -> alice( bob_free_day );
14       is_free@alice = isFreeDay( bob_free_day );
15     } prop { N.scope_name = "matching day" };
16     if( is_free )@alice {
17       scope @bob {
18         proposal: bob( "cinema" ) -> alice( event );
19         agreement@alice = getInput( "Bob proposes "  + event +
20           ", do you agree?[y/n]");
21         if( agreement == "y" )@alice{
22           end@bob = true;
23           book: bob( bob_free_day ) -> cinema( book_day );
24           ticket@cinema = getTicket( book_day );
25           { notify: cinema( ticket ) -> bob( ticket )
26             | notify: cinema( ticket ) -> alice( ticket ) }}
27       } prop { N.scope_name = "event selection" } };
28     if( !end )@bob {
29       _r@bob = getInput( "Alice refused. Try another date?[y/n]" );
30       if( _r != "y" )@bob{ end@bob = true }}}
```

Listing 1.1. Appointment program

The code starts with some deployment information (Lines 1-7), discussed later on. The behaviour starts at Line 9. The program is made by a cycle where Bob first checks when Alice is available and then invites her to the cinema. Before starting the cycle, Bob initialises the variable `end`, used in the guard of the cycle, to the boolean value `false` (Line 9). Note the annotation `@bob` meaning that `end` is local to Bob. The first instructions of the while are enclosed in a scope (Lines 11-15), meaning that they may be adapted in the future. The first operation within the scope is the call to the primitive function `getInput` that asks to Bob a day where he is free and stores this date into the local variable `free_day`. At Line 13 the content of `free_day` is sent to Alice via operation `proposal`. Alice stores it in its local variable `bob_free_day`. Then, at Line 14, Alice calls the external function `isFreeDay` that checks whether she is available on `bob_free_day`. If she is available (Line 16) then Bob sends to her the invitation to go to the cinema via the operation `proposal` (Line 18). Alice, reading from the input, accepts or refuses the invitation (Line 19). If Alice accepts then Bob first sets the variable `end` to `true` to end the cycle. Then, he sends to the cinema the booking request via operation `book`. The cinema generates the tickets using the external function `getTicket` and sends them to Alice and Bob via operation `notify`. The two notifications are done in parallel using the parallel operator `|` (until now we composed statements using the sequential operator ;). Lines 18-26 are enclosed in a second scope with property N.scope_name = "event selection". If the agreement is not reached, Bob decides, reading from the input, if he wants to stop inviting Alice. If so, the program exits.

We remark the different meanings of the annotations @bob and @alice. When prefixed by a variable, they identify the owner of the variable. Prefixed by the boolean guard of conditionals and cycles, they identify the role that evaluates the guard. Prefixed by the keyword scope, they identify the process coordinating the adaptation of that scope. A scope, besides the code, may also include some properties describing the current implementation. These can be specified using the keyword prop and are prefixed by N. For instance, each scope of the example includes the property scope_name, that can be used to distinguish its functionality.

AIOCJ can interact with external services, seen as functions. This allows both to interact with real services and to have easy access to libraries from other languages. To do that, one must specify the address and protocol used to interact with them. For instance, the external function isFreeDay used in Line 14 is associated to the service deployed at the domain "calendar.org", reachable though port 80, and that uses http as serialisation protocol (Line 1). External functions are declared with the keyword include. To preserve deadlock freedom, external services must be non-blocking. After function declaration, in a preamble section, it is possible to declare the locations where processes are deployed. The keyword starter is mandatory and defines which process must be started first. The starter makes sure all other processes are ready before the execution of the choreography begins.

Now suppose that Bob, during summer, prefers to invite Alice to a picnic more than to the cinema, provided that the weather forecasts are good. This can be obtained by adding the following adaptation rule to one of the adaptation servers. This may even be done while the application is running, e.g., while Bob is sending an invitation. In this case, if the first try of Bob is unsuccessful, in the second try he will propose a picnic.

```
1  rule {
2    include getWeather from "socket://localhost:8002"
3    on { N.scope_name == "event selection" and E.month > 5 and E.month < 10 }
4    do { forecasts@bob = getWeather( free_day );
5      if( forecasts == "Clear" )@bob{
6        eventProposal: bob( "picnic" ) -> alice( event )
7      } else { eventProposal: bob( "cinema" ) -> alice( event ) };
8      agreement@alice = getInput( "Bob proposes " + event +
9        ", do you agree?[y/n]");
10     if( agreement == "y" )@alice {
11       end@bob = true |
12       if( event == "cinema" )@alice {
13         //cinema tickets purchase procedure
14 }}}}
```

Listing 1.2. Event selection adaptation rule

A rule specifies its applicability condition and the new code to execute. In general, the applicability condition may depend only on properties of the scope, environment variables, and variables belonging to the coordinator of the scope. In this case, the condition, introduced by the keyword on (Line 3), makes the rule applicable to scopes having the property scope_name equal to the string "event selection" and only during summer. This last check relies on an environment variable month that contains the current month. Environment variables are prefixed by E.

When the rule applies, the new code to execute is defined using the keyword `do` (Line 4). In this case, the forecasts can be retrieved calling an external function `getWeather` (Line 4) that queries a weather forecasts service. This function is declared in Line 2. If the weather is clear, Bob proposes to Alice a picnic, the cinema otherwise. Booking (as in Listing 1.1, Lines 23-26) is needed only if Alice accepts the cinema proposal.

As detailed in [9], to obtain a deadlock-free application, we require the code of choreographies and rules to satisfy a well-formedness syntactic condition called *connectedness*. Intuitively, connectedness ensures that sequences of actions are executed in the correct order and avoids interference between parallel interactions. Requiring this condition does not hamper programmability, since it naturally holds in most of the cases, and it can always be enforced automatically via small patches to the choreography which preserve the behaviour of the program, as discussed in [17]. Also, checking connectedness is efficient, i.e., polynomial w.r.t. the size of the code [9].

3 Implementation

Our prototype implementation of AIOCJ is composed of two elements: the AIOCJ Integrated Development Environment (IDE), named AIOCJ-ecl, and the adaptation middleware that enables AIOC programs to adapt, called AIOCJ-mid.

AIOCJ-ecl is a plug-in for Eclipse [18] based on Xtext [19]. Xtext provides features such as syntax highlighting, syntax checking, and code completion, which help developers in writing choreographies and adaptation rules. Also, starting from a grammar, Xtext generates the parser for programs written in the AIOC language. Result of the parsing is an abstract syntax tree (AST) we use to implement (*i*) the checker for connectedness for choreographies and rules and (*ii*) the generation of Jolie code for each role. The connectedness check has polynomial computational complexity [9] thus making it efficient enough to be performed on-the-fly while editing the code.

The target language of code generation is Jolie [14]. Jolie supports architectural primitives such as dynamic embedding, aggregation, and redirection that we exploit to implement the adaptation mechanisms. Moreover, Jolie supports a wide range of communication technologies (TCP/IP sockets, local memory, Bluetooth) and of data formats (e.g., HTTP, SOAP, JSON). AIOCJ inherits this ability. The compilation generates a Jolie service for each role. The execution of scopes is delegated to sub-services accessed using Jolie redirection facility. Adaptation is enacted by disabling the current sub-service and replacing it with a new one, obtained from the adaptation server. To grant to all the sub-services access to variables, the state is stored by a dedicated sub-service local to the role. Auxiliary messages are exchanged to ensure that both the adaptation and the choices taken by the if and while constructs are done in a coordinated way. In particular, the scope execution not only requires interaction with the adaptation manager, but also communications among the different roles, ensuring that they all agree on whether adaptation is needed or not, and, in case, on which rule to apply. Indeed, the decision is taken by the role coordinating the adaptation and then communicated to other roles. Note that the different roles cannot autonomously take the decision, since if they take it at different times, changes in the environment or in the sets of available rules may lead to inconsistent decisions.

Synchronous message exchange is implemented on top of an asynchronous communication middleware by a sub-service that works as a *message handler*. The message handler of the starter role also ensures that, before the actual communication in the choreography starts, all the roles are ready.

AIOCJ-mid is implemented in Jolie and it includes:

- many, possibly distributed, *adaptation servers* where rules are published. Adaptation servers can be deployed and switched on and off at runtime;
- an *adaptation manager* that acts as a registry for adaptation servers and clients;
- an *environment* service that stores and makes available environment information. Environment information can change at any moment.

When an AIOCJ program reaches a scope, it queries the adaptation manager for a rule matching that scope. The adaptation manager queries each adaptation server sequentially, based on their order of registration. Each server checks the applicability condition of each of its rules. The first rule whose applicability condition holds is applied. In particular, the code of the rule is sent to the role coordinating the adaptation (via the adaptation manager) which distributes it to the involved roles. In each role, the new code replaces the old one. The study of more refined policies for rule selection, e.g., based on priorities, is a topic for future work.

4 Validation

In this section, we give a preliminary empirical validation of our implementation. The main aim is to test how our mechanisms for adaptation impact on performances.

In the literature, to the best of our knowledge, there is no approach to adaptation based on choreography programming. Thus, it is difficult to directly compare our results with other existing approaches. Moreover, we are not aware of any established benchmark to evaluate adaptive applications. For this reason, we tested AIOCJ performances by applying it to two typical programming patterns: *pipes* and *fork-joins*. Since we are interested in studying the cost of adaptation, our scenarios contain minimal computation and are particularly affected by the overhead of the adaptation process. Clearly, the percentage of the overhead due to adaptation will be far lower in real scenarios, which are usually more computationally intensive. In the first scenario, we program a pipe executing n tasks (in a pipe, the output of task t_i is given as input to task t_{i+1}, for $i \in \{1, \dots, n-1\}$). To keep computation to a minimum, each task simply computes the increment function. In the fork-join scenario, n tasks are computed in parallel. Each task processes one character of a message of length n, shifting it by one position. The message is stored in an external service.[1]

To enable adaptation, each task is enclosed in a scope. We test both scenarios with an increasing number of tasks $n \in \{10, 20, \dots, 100\}$ to study how performances scale as the number of adaptation scopes increases. We evaluate performances in different contexts, thus allowing us to understand the impact of different adaptation features, such as scopes, adaptation servers, and adaptation rules.

[1] The code of both scenarios is in the companion technical report [16].

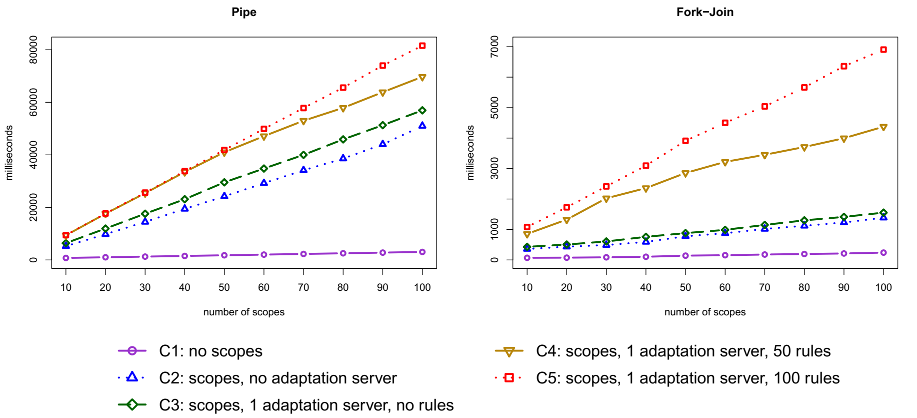

Fig. 1. Times of execution of the pipe (left) and the fork-join (right) scenarios

Context 1: no scopes, no adaptation servers, no rules;
Context 2: each task is enclosed in a scope, no adaptation servers, no rules;
Context 3: each task is enclosed in a scope, one adaptation server, no rules;
Context 4: as Context 3, but now the adaptation server contains 50 rules. Each rule is applicable to a unique scope i, and no rule is applicable to scopes with $i > 50$. The rules are stored in random order.
Context 5: as Context 4, but with 100 rules, one for each scope.

Each rule in Contexts 4 and 5 is applicable to one specific scope only (through a unique property of the scope), hence when testing for 50 rules, only the first 50 scopes adapt.

We repeated every test 5 times. We performed our tests on a machine equipped with a 2.6GHz quad-core Intel Core i7 processor and 16GB RAM. The machine runs Mavericks 10.9.3, Java 1.7.55, and Jolie r.2728. Figure 1 shows the tests for the pipe (left) and the fork-join (right). Both charts display on the x-axis the number of tasks/scopes and on the y-axis the execution time in milliseconds.

As expected, in both scenarios there is a significant gap between Contexts 1 and 2. In words, the introduction of scopes has a strong effect on performances. The ratio is 1:13 for the pipe scenario and 1:5.5 for the fork-join scenario. This is due to the auxiliary communications needed to correctly execute a scope. The observed overhead is higher in the pipe scenario, since different scopes check for adaptation in sequence, while this is done in parallel for the fork-join scenario.

Adding an adaptation server (from Context 2 to Context 3) has little impact on performances: 19% of decay for pipe, and 17% for fork-join. The figures are reasonable, considered that Context 3 adds only one communication w.r.t. Context 2.

On the contrary, there is a notable difference when adding rules to the adaptation server (Context 4 is 1.4 times slower than Context 3 for the pipe scenario, 2.9 for the fork-join scenario). In Contexts 4 and 5, performances are really close up to 50 scopes (in the pipe scenario they almost overlap) although Context 5 has twice the rules of Context 4. This illustrates that the time to test for applicability of rules is negligible. Hence, the highest toll on performances is related to actual adaptation, since it requires to transfer and embed the new code. This is particularly evident in the fork-join scenario where multiple adaptations are executed in parallel and the adaptation server becomes a bottleneck. This problem can be mitigated using multiple distributed adaptation servers.

The fact that the most expensive operations are scope execution and actual adaptation is highlighted also by the results below. The table shows the cost of different primitives, including scopes in different contexts. Times are referred to 5 executions of the sample code in the companion technical report [16].

Test	Time (ms)	Test	Time (ms)
assignment	**2.2**	scope, 1 adaptation server, 1 matching rule	**280.6**
interaction	**4.2**	scope, 1 adaptation server, 50 rules, none matching	**254.2**
if statement	**16.6**	scope, 1 adaptation server, 50 rules, 1 matching	**338.6**
scope, no adaptation server	**129.4**	scope, 1 adaptation server, 100 rules, none matching	**310.2**
scope, 1 adaptation server, no rule	**203.8**	scope, 1 adaptation server, 100 rules, 1 matching	**385**

As future work we will exploit these results to increase the performances of our framework, concentrating on the bottlenecks highlighted above. For instance, scope execution (as well as conditionals and cycles) currently requires many auxiliary communications ensuring that all the processes agree on the chosen path. In many cases, some of these communications are not needed, since a process will eventually discover the chosen path from the protocol communications. Static analysis can discover redundant communications and remove them. Another improvement is letting the adaptation server send the new code directly to the involved roles, skipping the current forward chain.

5 Related Work and Conclusion

This paper presented a framework for programming rule-based adaptation of distributed applications. Its distinctive trait is that, being based on a choreographic approach, it guarantees deadlock-freedom by construction for the running distributed application, even in presence of adaptation rules which were unknown when the application was started, and for any environment condition.

Adaptation is a hot topic, and indeed there is a plethora of approaches in the literature, see, e.g., the surveys [20, 21]. However, approaches based on formal methods are only emerging recently and few of them have been implemented in a working tool. In particular, the use of choreographies to capture and define adaptive applications is a novel idea. For a discussion of works on adaptation with formal bases, but which have not been implemented, we refer to [9]. Here, we just recall [22], which exploits a choreographic approach for self-adaptive monitoring of distributed applications.

Among the implemented approaches, the most related to ours is JoRBA [5]. JoRBA features scopes and adaptation rules similar to ours. However, JoRBA applications are not distributed and JoRBA does not guarantee any property of the adapted application.

In [23] choreographies are used to propagate protocol changes to the other peers, while [24] presents a test to check whether a set of peers obtained from a choreography can be reconfigured to match a second one. Differently from ours, these works only provide change recommendations for adding and removing message sequences.

Various tools [25–27] exploit automatic planning techniques in order to elaborate, at runtime, the best sequence of activities to achieve a given goal. These techniques are more declarative than ours, but, to the best of our knowledge, they are not guaranteed to always find a plan to adapt the application.

Among the non-adaptive languages, Chor [2] is the closest to ours. Indeed, like ours, Chor is a choreographic language that compiles to Jolie. Actually, AIOCJ shares part of the Chor code base. However, due to the different semantics of the sequential operator and the lack of the parallel composition in Chor, a faithful encoding of the scenarios in Section 4 is not possible, especially for the fork-join scenario. On an almost equivalent implementation of the pipe scenario, Chor proves to be more efficient than AIOCJ.

In the future, we would like to test the expressive power of our language, trying to encode patterns of adaptation from existing approaches. An obvious benefit of such an encoding is that it will capture patterns of adaptation used in real-case scenarios, guaranteeing also deadlock freedom, which is not provided by other approaches. This task is cumbersome, due to the huge number and heterogeneity of those approaches. Nevertheless, we already started it. In particular, in the website [28], we show how to encode examples coming from distributed [29] and dynamic [30] Aspect-Oriented Programming (AOP) and from Context-Oriented Programming (COP) [31]. In general, we can deal with cross-cutting concerns like logging and authentication, typical of AOP, viewing point-cuts as empty scopes and advices as adaptation rules. Layers, typical of COP, can instead be defined by adaptation rules which can fire according to contextual conditions captured by the environment. Possible extensions of our framework include the use of asynchronous communications in the AIOC language and the introduction of mechanisms to deal with exceptions and failures. Finally, we would like to pursue a systematic analysis of the workflow change patterns like the ones presented in [32, 33], showing how these patterns are captured by AIOCJ.

References

1. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS, pp. 329–339. ACM (2008)
2. Carbone, M., Montesi, F.: Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In: POPL, pp. 263–274. ACM (2013)
3. Gößler, G., Sifakis, J.: Component-Based Construction of Deadlock-Free Systems. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 420–433. Springer, Heidelberg (2003)
4. Naik, M., Park, C.-S., Sen, K., Gay, D.: Effective static deadlock detection. In: ICSE, pp. 386–396. IEEE (2009)
5. Lanese, I., Bucchiarone, A., Montesi, F.: A Framework for Rule-Based Dynamic Adaptation. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 284–300. Springer, Heidelberg (2010)
6. Bucchiarone, A., Marconi, A., Pistore, M., Raik, H.: Dynamic Adaptation of Fragment-Based and Context-Aware Business Processes. In: ICWS, pp. 33–41. IEEE Press (2012)
7. Chen, W.-K., Hiltunen, M.A., Schlichting, R.D.: Constructing Adaptive Software in Distributed Systems. In: ICDCS. LNCS, vol. 6084, pp. 635–643. Springer (2001)
8. Zhang, J., Goldsby, H., Cheng, B.H.C.: Modular Verification of Dynamically Adaptive Systems. In: AOSD, pp. 161–172. ACM (2009)

9. Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Deadlock Freedom by Construction for Distributed Adaptative Applications. Technical Report, `http://arxiv.org/pdf/1407.0970v1.pdf`
10. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. 34(2), 8 (2012)
11. Scribble website, `http://www.jboss.org/scribble`
12. Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the Gap between Interaction- and Process-Oriented Choreographies. In: SEFM, pp. 323–332. IEEE Press (2008)
13. World Wide Web Consortium, Web Services Choreography Description Language Version 1.0 (2005), `http://www.w3.org/TR/ws-cdl-10/`
14. Jolie website, `http://www.jolie-lang.org/`
15. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Proc. of ECOWS 2007, pp. 13–22. IEEE Press (2007)
16. Dalla Preda, M., Giallorenzo, S., Lanese, I., Mauro, J., Gabbrielli, M.: AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications. Technical Report, `http://arxiv.org/pdf/1407.0975.pdf`
17. Lanese, I., Montesi, F., Zavattaro, G.: Amending choreographies. In: WWV, vol. 123, pp. 34–48. EPTCS (2013)
18. Eclipse website, `http://www.eclipse.org/`
19. Xtext website, `http://www.eclipse.org/Xtext/`
20. Ghezzi, C., Pradella, M., Salvaneschi, G.: An evaluation of the adaptation capabilities in programming languages. In: SEAMS, pp. 50–59. ACM (2011)
21. Leite, L.A.F., et al.: A systematic literature review of service choreography adaptation. Service Oriented Computing and Applications 7(3), 199–216 (2013)
22. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Self-adaptive monitors for multiparty sessions. In: PDP, pp. 688–696. IEEE (2014)
23. Rinderle, S., Wombacher, A., Reichert, M.: Evolution of Process Choreographies in DYCHOR. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 273–290. Springer, Heidelberg (2006)
24. Wombacher, A.: Alignment of choreography changes in BPEL processes. In: IEEE SCC, pp. 1–8. IEEE Press (2009)
25. Cugola, G., Ghezzi, C., Pinto, L.S.: DSOL: a declarative approach to self-adaptive service orchestrations. Computing 94(7), 579–617 (2012)
26. Baresi, L., Marconi, A., Pistore, M., Sirbu, A.: Corrective Evolution of Adaptable Process Models. In: Nurcan, S., Proper, H.A., Soffer, P., Krogstie, J., Schmidt, R., Halpin, T., Bider, I. (eds.) RIMS 1982. LNBIP, vol. 147, pp. 214–229. Springer, Heidelberg (2013)
27. Bucchiarone, A., Marconi, A., Mezzina, C.A., Pistore, M., Raik, H.: On-the-Fly Adaptation of Dynamic Service-Based Systems: Incrementality, Reduction and Reuse. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 146–161. Springer, Heidelberg (2013)
28. AIOCJ website, `http://www.cs.unibo.it/projects/jolie/aiocj.html`
29. Pawlak, R., et al.: JAC: an aspect-based distributed dynamic framework. SPE 34(12), 1119–1148 (2004)
30. Yang, Z., Cheng, B.H.C., Stirewalt, R.E.K., Sowell, J., Sadjadi, S.M., McKinley, P.K.: An aspect-oriented approach to dynamic adaptation. In: WOSS, pp. 85–92. ACM (2002)
31. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology 7(3), 125–151 (2008)
32. Weber, B., Rinderle, S., Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)
33. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow Evolution. Data Knowl. Eng. 24(3), 211–238 (1998)