Developing correct, distributed, adaptive software

Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro

Department of Computer Science, University of Bologna/ Lab. Focus INRIA, Italy

Abstract

We illustrate our approach to develop and verify distributed, adaptive software systems. The cornerstone of our framework is the use of choreography languages, which allow us to obtain correctness by construction. Behavioural Design Patterns are also used as abstract tools to design real systems, while techniques based on abstract interpretation and on dynamic verification are integrated in our framework to reduce the complexity of verification.

Keywords: Choreography languages, Abstract Interpretation, Adaptation.

1. Introduction

Adaptive, distributed software has applications in many domains and systems, exhibiting deeply different characteristics. Long running (often distributed) systems live for long periods of time and therefore should adapt to varying contextual conditions, user requirements and execution environments. More importantly, the details of the adaptation needs, and the solutions to be used to answer them, may not be known when the system is designed, deployed or even started. Such systems thus need to dynamically adapt their behavior (this can be autonomic, or may require an external intervention). Particularly important in this context are mission critical Adaptive Control Systems used in Cyber-Physical systems to respond to changes in the physical environment.

Development and verification of distributed, adaptive systems pose several formidable challenges. First, the current development technology is not well suited to develop and verify large-scale adaptive distributed systems due to the lack of high-level structuring abstractions for complex communication behaviour or for context-aware adaptation. In recent years, session types, choreography languages, and behavioural contracts have been advocated as possible abstractions for providing high-level specifications describing the expected behaviour of a distributed system [8, 19, 9, 25, 10, 1]. These concepts usually consider static techniques, which alone are not sufficient to model dynamically adaptable software. Indeed, the assumption on either the ability to type-check the component source code, or the availability of its complete behavioural interface, may not be realistic in presence of adaptation. Particularly relevant in

this context are Interaction-Oriented Choreographies (IOCs) [25] which allow to abstractly describe the participants to a distributed protocol, the interactions among them, and their order. From an IOC one can automatically generate a detailed description of the behaviour of each participant, expressed in terms of a *Process-Oriented Choreography* (POC), which, in many cases, provides an executable code. A main result in this setting is that the POC automatically derived from a given IOC correctly implements the behaviour specified by the given IOC, inheriting relevant correctness properties such as deadlock freedom [25]. IOCs and their projection onto POCs have been studied in different contexts [25, 8, 19, 10], and integrated into different kinds of languages [29, 9]. A relevant limitation affecting all the IOC-based approaches is that they can be applied only to systems whose structure is static and fully known since the very beginning.

Concerning more specifically adaptive systems, several middlewares and architectures enabling run-time adaptation have been proposed in the literature, such as [5, 11, 24, 31, 17] (an interesting survey can be found in [26]). While these approaches provide tools for programming adaptive systems, the challenge of ensuring that those systems behave as expected after the execution of some adaptation steps is still open. One cannot know a priori the structure of the adapted system, and this seems to make the use of static analysis techniques impossible. For this reason, most of the approaches in the literature offer no guarantee on the behaviour of the adaptive system after adaptation [24, 5, 11], or they assume to know all the possible adaptations in advance [31].

Verification of adaptive systems is very difficult also because of the combinatorial effect due to the composition of different variants of the individual modules and to the run-time nature of system configuration. Many aspects of such composition and configuration can only be partially foreseen at design and compile time, hence performing static checks would require to consider a very large number of possibilities, including many which will never happen at run-time. A possible solution here could be to move a part of the static analysis to execution time, along the lines of what happens, for example, with the analysis of bytecode that the JVM performs at class loading time.

1.1. Our approach

In order to attack the challenges mentioned above we intend to develop effective methodologies and tools for proving correctness of adaptive software systems. Our approach is based on the integration of different techniques, including abstract interpretation, choreography languages, and design patterns. More precisely, we aim to reach the following objectives:

1. To define a framework which allows us to statically prove the correctness of adaptive distributed systems by using choreography languages.
2. To devise suitable design patterns which, exploiting the correctness results of our framework, allow us to construct correct, distributed, adaptive software.

3. To include in such a framework suitable abstract interpretation techniques to reduce the complexity of verification of real systems.
4. To develop a formal theory based on choreography languages and abstract interpretation for statically proving security properties of adaptive systems (e.g., non interference).
5. To integrate the static techniques with run-time monitoring and, more generally, dynamic verification techniques.

The rest of this paper is devoted to illustrate in more detail these objectives. For some of them, notably the first and the second, we have already several results [13, 15], while for others our research is at an early stage and we expect to have results in the medium-long term.

2. Correctness by construction: choreography languages

Correctness properties of adaptive systems can be imposed *by design*, by using choreography languages. Correctness by design is obtained by automatically deriving executable code for adaptive distributed systems from high level, IOC-like specifications. Our technique [13] is based on a careful split of the system specification into a description of the initial system, to be checked before deployment, and a description of the adaptation steps, each of which can be defined and checked in isolation, even while the system is running.

We use a *rule-based approach* to adaptation which relies on the following architectural model: the adaptive system is composed by interacting participants deployed on different localities, each executing its own code and accessing its own local state. Adaptation is performed by an *adaptation middleware*. The middleware includes one or more, possibly distributed, *adaptation servers*, which are repositories of *adaptation rules*. Adaptation rules can be added or removed at any moment, while the system is running. The running system may interact with the adaptation middleware to look for applicable adaptation rules. Among the different mechanisms proposed for adaptation, we concentrated on a simple yet powerful one: the possibility of replacing a predefined code region (possibly distributed among different participants) with new code tackling the new requirements. Applicability depends on conditions on the execution environment (possibly including user desires) and on properties of the code region to be replaced.

The syntax of *Adaptive IOC (AIOC) processes*, ranged over by $\mathcal{I}, \mathcal{I}', \ldots$, is defined as follows:

$$\begin{array}{rcl} \mathcal{I} & ::= & o : r_1(e) \to r_2(x) \mid \mathcal{I}; \mathcal{I}' \mid \mathcal{I} | \mathcal{I}' \mid x@r = e \\ & & \mathit{if}\ b@r\ \{\mathcal{I}\}\ \mathit{else}\ \{\mathcal{I}'\} \mid \mathit{while}\ b@r\ \{\mathcal{I}\} \\ & & \mathit{scope}\ l@r\ \{\mathcal{I}\} \end{array}$$

Interaction $o : r_1(e) \to r_2(x)$ means that the participant r_1 sends a message on operation o to participant r_2. The sent value is obtained by evaluating expression e in the local state of r_1. As a result of the communication, the value

is stored in local variable x in r_2. Processes $\mathcal{I};\mathcal{I}'$ and $\mathcal{I}|\mathcal{I}'$ denote sequential and parallel composition of $\mathcal{I}$ and $\mathcal{I}'$, respectively. Assignment $x@r = e$ assigns the result of the evaluation of expression e in the local state of participant r to its local variable x. Choice *if* $b@r$ $\{\mathcal{I}\}$ *else* $\{\mathcal{I}'\}$ executes process $\mathcal{I}$ if the evaluation of boolean expression b is `true` in the local state of r, process $\mathcal{I}'$ otherwise. Cycles are defined using *while* $b@r$ $\{\mathcal{I}\}$, which executes process $\mathcal{I}$ while the boolean expression b is `true` in the local state of r. The last construct is *scope* $l@r$ $\{\mathcal{I}\}$, that delimits a region $\mathcal{I}$ of the IOC process that may be adapted in the future. In *scope* $l@r$ $\{\mathcal{I}\}$, participant r coordinates the adaptation procedure by interacting with the adaptation middleware to check whether adaptation is needed. Also, l is the label of the scope, to be matched by a corresponding label in the adaptation rule.

As an example, consider the scenario where two buyers want to buy a book sharing half of the price of the book.

```
1  book_title@buyer1 = getInput( "Insert book");
2  proposal: buyer1( book_title ) -> buyer2( book_title );
3  answer@buyer2 = getInput( "Are you interested in buying " + book_title + "?");
4  agreement: buyer2( answer ) -> buyer1( answer );
5  if( answer == True )@buyer1{
6    scope pay@seller {
7      quote: buyer1( book_title ) -> seller( book_title );
8      price@seller = getPrice( book_title );
9      price: seller( price ) -> buyer1( price );
10     price: buyer1( price ) -> buyer2( price );
11     ... // payment procedure
12   }
13 }
```

Listing 1: AIOC for Two Buyers Scenario

Here, `buyer1` reads (using function `getInput()`) from the user the name of a book she is interested in into local variable `book_tile`. Then she sends the name of the book to `buyer2` via operation `proposal`. Then `buyer2` reads if the user is interested in the book and communicates the answer to `buyer1` via operation `answer`. In case of a positive answer, `buyer1` asks to the `seller` the price of the book that, upon computation via `getPrice` function, is sent back to `buyer1`. To start the payment procedure (here skipped for the sake of presentation) `buyer1` notifies the price of the book to `buyer2`. The code between Lines 7 and 11 is inserted in a scope with label `pay` which specifies that these instructions may, at some point during execution, be adapted to take into account new requirements. In this case, the `seller` is the participant responsible to interact with the adaptation middleware.

An adaptation step is described by rules having the following syntax

$$\textit{rule } l \textit{ where } \mathcal{C} \textit{ specifies } \mathcal{I}$$

where l is the label of the scopes to which the rule applies, $\mathcal{C}$ is a boolean predicate that specifies the applicability condition to be satisfied, and $\mathcal{I}$ is the AIOC process that will replace the scope in case the adaptation is performed. If a rule is applied, it replaces the code region of the distributed participants with a newer version, able to better meet the requirements. Adaptation of different participants is coordinated ensuring coherent behaviour.

For instance let us suppose that instead of splitting the price in half the administrator of the system always allows `buyer2` to agree on the share she is willing to spend. The rule implementing this adaptation can be specified as follows.

```
1  rule pay {
2    where { True }
3    specifies {
4      quote: buyer1( book_title ) -> seller( book_title );
5      price@seller = getPrice( book_title );
6      { price: seller( price ) -> buyer1( price ) |
7        price: seller( price ) -> buyer2( price ) };
8      agreement@buyer1 = false; continue@buyer1 = true;
9      while( ( not agreement ) and continue )@buyer1 {
10       share@buyer2 = getInput( "Insert share for " + book_title );
11       offer: buyer2( share ) -> buyer1( share );
12       agreement@buyer1 = getInput( "Offer: " + share + ", do you accept?" );
13       if( agreement )@buyer1 {
14         continue@buyer1 = getInput( "Do you want to ask for another offer?" )
15       }};
16     if ( agreement )@buyer1 {
17       ... // payment procedure
18     }}
```

Listing 2: Adaptation Rule for Two Buyers Scenario

This rule applies to scopes labeled `pay` every time the scope is reached (the applicability condition defined in Line 2 is indeed always true). In the new code `buyer1` asks to the seller the price of the book that is sent in parallel by the seller to both the buyers. Then `buyer2` enters the share she wants to pay and sends it to `buyer1`, that could accept or refuse the offer. In case of refusal `buyer1` can ask to `buyer2` to make another offer or withdraw the acquisition of the book. In case an agreement is reached the payment procedure, here omitted, is executed.

From an AIOC specification of a system one can automatically derive an (executable) distributed application by means of a projection function. A compositionality result and a protocol ensuring that adaptation is applied in a coordinated way allow us to prove relevant properties of the adapted system. Provided that an AIOC and the rules satisfy some simple syntactic conditions, we have as a theorem that the choreographic specification has the same behaviour of the projected and distributed programs. As a corollary, for all possible adaptation scenarios, we can ensure properties such as deadlock freedom or termination of the projected code.

It is worth noticing that, even though adaptation steps can occur at run-time, these checks are all static: in order to ensure correctness the initial systems and the rules describing the adaptation steps have to satisfy some syntactic conditions which do not require run-time information.

To validate this approach on real, executable case studies, we have implemented a framework where AIOC are projected on code written in Jolie [20, 27], an open-source service-oriented programming language.

2.1. Behavioural Design Patterns

Design patterns [16] define generic reusable solutions to recurring problems and are largely used in the software engineering practice for building complex

systems. *Workflow Patterns* [30] were defined as patterns that describe recurring flow of interactions among several participants. They are independent and composable, following the basic design principle of Service Oriented Computing where a new service is defined as a composition of other existing one. In [15] many workflow patterns have been implemented in Jolie [20], thus providing a first step towards the realization of a library that programmers can use for realizing Service Oriented Architectures (SOA) based on Workflow Patterns.

More generally, workflow patterns could be used to build real SOAs and distributed systems ensuring (partial) correctness by construction, along the lines of what has been discussed in this section. The basic idea here is to use the AIOC language to express specific Behavioural Design Patterns (BDP) [15], which act as a sort of templates were specific parts can be left unspecified. The correctness results on the AIOC, together with suitable conditions, ensure that the relevant correctness properties of the system are preserved, once the "holes" in the templates are "filled" by specific, simple pieces of code (which satisfy the conditions). Also, in case of adaptation, these parts could be replaced by others while maintaining correctness. The interesting point is that these abstract BDPs can be automatically translated into an executable language (for example Jolie) by using the projection function: for each BDP described in terms of IOC we can derive Jolie code for the behaviour of each participant. For space reason we omit to describe the existing BDPs, however Listings 1 and 2 provide some examples: basic patterns like the Sequence and the Parallel Split patterns are provided respectively by the sequence (`;`) and the parallel (`|`) operators, while Lines 6-7 in Listing 2 contain a Synchronisation pattern.

3. Further tools for correctness

The approach outlined in the previous section does not allow to prove all relevant properties of a system, hence further analysis and verification phases may be needed, both static and dynamic. In this section we examine the most relevant ones, namely static analysis by abstract interpretation and monitoring.

3.1. Abstract intrepretation

When considering analysis and verification by using abstract interpretation the presence of adaptation causes several difficulties. In fact, static analysis of real systems is usually performed in a modular way which requires to specify, for each method, pre- and post-conditions and, for each class, an object invariant[1]. In the context of adaptive systems this approach is impractical, since the presence of many different objects, which may interact in complex and even unpredictable ways, makes the system difficult to describe and the analysis a daunting task: considering all possible interactions and behaviours, including those which will never happen in the deployed system, could easily

[1]We are referring here to object oriented languages. The case of other programming paradigms is analogous.

produce a combinatorial explosion. A key point to tackle these problems is to define suitable abstract semantics able to capture the structure of programs that change at run-time, taking inspiration from the techniques used for modelling self-modifying/metamorphic malware in [12].

We also intend to move part of the static analysis at execution time: when components are loaded (or generated) at run-time, the entire system is known, so that properties may be checked at global level.

We plan to use abstract interpretation techniques also to verify security properties of adaptive systems. Some recent works have proposed to use session types for the verification of security properties such as integrity [2, 28], access control [22], and information flow [7, 6]. These analyses can be extended to adaptive systems by defining a notion of *secure AIOC* (for example an AIOC with no information leakage) and then identifying the constraints that the AIOC, the adaptation rules, and the projection function have to satisfy in order to ensure that the projected code still satisfies the non-interference security requirements. More specifically, we plan to consider the generalized version of non-interference known as Abstract Non-Interference (ANI) [18], where secret and public properties are modelled as abstractions. ANI can be expressed in terms of completeness in Abstract Interpretation (a well studied property of abstractions which can be systematically enforced using the completeness refinement technique). We believe that the relation between the projected code and AIOC could be formalized as an abstraction in the abstract interpretation framework. Then, the satisfaction of the ANI properties of the AIOC could be expressed at the projected code level as a completeness problem, and hence the completeness refinement could be used to derive a strategy for enforcing ANI at the projected code level.

3.2. Monitoring and dynamic verification

As previously discussed, choreography specifications are used to generate concrete services via projection. The performances of this automatically generated code may however be worse than the ones of ad-hoc developed services. For this reason, one may manually optimize some of the services and this may break correctness by design. Thus one should verify correctness at run-time by using monitors and other dynamic techniques to check whether the system behaves correctly. For example, one could be interested in checking, locally for every service, the arrival order of the messages, while a relevant global property to check is that all the participants of a payment transaction receive the (success or failure) payment notification.

The presence of a choreographic description can be useful in this case since monitors used to check the validity of local and global proprieties can be derived automatically by exploiting the choreography. Indeed, choreographic specifications, possibly enhanced with assertions or constraints to express proprieties not captured by the choreographic description (e.g., [3]), could be used to create, for every service, a local monitor which uses the incoming and outgoing message flow to check local proprieties. Global proprieties could instead be checked by

a global monitor, still derived from the choreography, that collects a subset of the logs of the local monitors to verify if the global property is violated.

Following [4], this approach could be implemented by first deriving from the choreography specification suitable monitoring rules, which could be expressed in formalism like the Event Calculus [23]. These rules can then be effectively checked by using business rule management systems like Drools [14]. Potential violations of the choreography specification could then be signalled to the administrator and culprit detection mechanisms could be adopted in order to check which participant was responsible for it. More generally, suitable dynamic verification techniques could be integrated in order to enhance the verification capabilities of our framework.

4. Conclusion

We have illustrated our approach to develop and verify distributed, adaptive software systems. The approach is intended to be used for building real systems, indeed we have already implemented the projection function which from high level adaptive IOC specifications produces executable Jolie code which is correct by construction, as it inherits all the properties of the IOC level.

This approach to correctness by construction, further detailed in [13], is the first one which tries to integrate adaptation techniques into choreography-based languages. Such an integration, in our opinion, can foster relevant results, since IOCs and their projections onto executable languages are important tools for specifying and programming correct, complex distributed systems. This approach is further enhanced by defining at IOC level suitable Behavioural Design Patterns, which, as previously discussed, can be used by a programmer as templates for designing correct software. Several possible improvement are possible, in particular the matching of rules with scopes for performing adaptation steps is currently based only on labels. One can easily imagine to use also more refined techniques based on preconditions and postconditions specified in a suitable assertion language, and even to exploit information provided by specific ontologies, in order to express more sophisticated matching policies. To develop all these extensions and check the syntactical properties needed by the AIOC we are considering the use of Rascal [21], a DSL language framework that allows an high level integration of source code analysis and manipulation.

Also the integration of choreography languages with abstract interpretation and dynamic techniques is particularly important for addressing real systems, as it allows to reduce the complexity in verification arising from adaptation. In this case we have only preliminary results, and most of the work has to be done in the near future.

References

[1] Basu, S., Bultan, T., Ouederni, M., 2012. Deciding choreography realizability, in: Proc. of POPL 2012, ACM. pp. 191–202.

[2] Bhargavan, K., Corin, R., Deniélou, P.M., Fournet, C., Leifer, J.J., 2009. Cryptographic protocol synthesis and verification for multiparty sessions, in: Proc. of the 2009 22nd IEEE CSFS, IEEE CS. pp. 124–140.

[3] Bocchi, L., Chen, T.C., Demangeon, R., Honda, K., Yoshida, N., 2013. Monitoring Networks through Multiparty Session Types, in: FMOODS/-FORTE, Springer Berlin Heidelberg. pp. 50–65.

[4] Bragaglia, S., Chesani, F., Mello, P., Sottara, D., 2012. A Rule-Based Calculus and Processing of Complex Events, in: RuleML, pp. 151–166.

[5] Bucchiarone, A., Marconi, A., Pistore, M., Raik, H., 2012. Dynamic Adaptation of Fragment-Based and Context-Aware Business Processes, in: Proc. of ICWS 2012, IEEE Press. pp. 33–41.

[6] Capecchi, S., Castellani, I., Dezani-Ciancaglini, M., 2011. Information flow safety in multiparty sessions, in: Proc. 18th EXPRESS, pp. 16–30.

[7] Capecchi, S., Castellani, I., Dezani-Ciancaglini, M., Rezk, T., 2010. Session types for access and information flow control, in: Concurrency Theory, 21th International Conference, CONCUR 2010, Springer. pp. 237–252.

[8] Carbone, M., Honda, K., Yoshida, N., 2012. Structured communication-centered programming for web services. ACM TPLS 34, 8.

[9] Carbone, M., Montesi, F., 2013. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming, in: Proc. of POPL 2013, ACM. pp. 263–274.

[10] Castagna, G., Dezani-Ciancaglini, M., Padovani, L., 2012. On global types and multi-party session. Logical Methods in Computer Science 8.

[11] Chen, W.K., Hiltunen, M.A., Schlichting, R.D., 2001. Constructing Adaptive Software in Distributed Systems, in: Proc. of ICDCS'01, Springer. pp. 635–643.

[12] Dalla Preda, M., Giacobazzi, R., Debray, S.K., Coogan, K., Townsend, G.M., 2010. Modelling metamorphism by abstract interpretation, in: Proc. of SAS10, Springer-Verlag. pp. 218–235.

[13] Dalla Preda, M., Lanese, I., Mauro, J., Gabbrielli, M., Giallorenzo, S., 2013. Safe Run-time Adaptation of Distributed Systems. Technical Report. `http://www.cs.unibo.it/~sgiallor/papers/SRAoDS.pdf`.

[14] Drools, 2013. Drools website. URL: `http://www.jboss.org/drools`.

[15] Gabbrielli, M., Giallorenzo, S., Montesi, F., 2013. Workflow Patterns for Coordinating Heterogeneous Networks. Technical Report. `http://www.cs.unibo.it/~sgiallor/papers/WPfCHN_full_paper.pdf`.

[16] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[17] Ghezzi, C., Pradella, M., Salvaneschi, G., 2011. An Evaluation of the Adaptation Capabilities in Programming Languages, in: Proc. of SEAMS 2011, ACM. pp. 50–59.

[18] Giacobazzi, R., Mastroeni, I., 2004. Abstract non-interference: parameterizing non-interference by abstract interpretation, in: Proc. ACM SIGPLAN-SIGACT, POPL 2004, pp. 186–197.

[19] Honda, K., Yoshida, N., Carbone, M., 2008. Multiparty Asynchronous Session Types, in: Proc. of POPL'08, ACM Press. pp. 273–284.

[20] Jolie, 2013. Jolie website. URL: `http://www.jolie-lang.org/`.

[21] Klint, P., van der Storm, T., Vinju, J.J., 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation, in: SCAM, IEEE Computer Society. pp. 168–177.

[22] Kolundžija, M., 2009. Web services and formal methods, Springer-Verlag. chapter Security Types for Sessions and Pipelines, pp. 175–190.

[23] Kowalski, R., Sergot, M., 1986. A logic-based calculus of events. New Gen. Comput. 4, 67–95.

[24] Lanese, I., Bucchiarone, A., Montesi, F., 2010. A Framework for Rule-Based Dynamic Adaptation, in: Proc. of TGC 2010, Springer. pp. 284–300.

[25] Lanese, I., Guidi, C., Montesi, F., Zavattaro, G., 2008. Bridging the Gap between Interaction- and Process-Oriented Choreographies, in: Proc. of SEFM'08, IEEE Press. pp. 323–332.

[26] Leite, L., Ansaldi Oliva, G., Nogueira, G., Gerosa, M., Kon, F., Milojicic, D., 2012. A systematic literature review of service choreography adaptation. Service Oriented Computing and Applications , 1–18.

[27] Montesi, F., Guidi, C., Zavattaro, G., 2007. Composing services with JOLIE, in: Proc. of ECOWS'07, IEEE Press. pp. 13–22.

[28] Planul, J., Corin, R., Fournet, C., 2009. Secure enforcement for global process specifications, in: Proc. of CONCUR 2009, Springer-Verlag. pp. 511–526.

[29] Scribble, 2013. Scribble website. URL: `http://www.jboss.org/scribble`.

[30] Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P., 2003. Workflow patterns. Distrib. Parallel Databases 14, 5–51.

[31] Zhang, J., Goldsby, H., Cheng, B.H.C., 2009. Modular Verification of Dynamically Adaptive Systems, in: Proc. of AOSD'09, ACM. pp. 161–172.