FoCUS, on Implementations of Service-Oriented Computing

Saverio Giallorenzo
FoCUS | A Positive Loop

Theory
Make problems tractable
Formalise new ideas

Implementation
Real-world problems
New technologies
A case of a Service-Oriented Programming Language
**SOCK: A Calculus for Service Oriented Computing**

Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro

---

**Abstract.**
Service oriented computing is an emerging paradigm for designing distributed applications where service and composition are the main concepts it is based upon. In this paper we propose SOCK, a three-layered calculus equipped with a formal semantics, for addressing all the basic mechanisms of service communication and composition. The main contribute of our work is the development of a formal framework where the service design is decomposed into three fundamental parts: the behaviour, the declaration and the composition where each part can be designed independently of the other ones.

1 Introduction

Service oriented computing (SOC) is an emerging paradigm for designing distributed applications where service and composition are the main concepts it is based upon. A service can be seen as an application which performs a certain task when it is invoked. A composition of services can be seen as a group of services that, by means of collaborating message exchanges, fulfills a more complex task than those performed by the single services it is composed of. The key fact is that a suitable composition of services is a service. The most credited service oriented technology is the Web Services. A lot of industries and consortia in the world like Microsoft, IBM, W3C, OASIS (just to mention a few) have developed standards which define Web Services interfaces such as WSDL [Wor] and composition languages such as WS-BPEL [OAS]. Such a kind of languages are based on XML and are not equipped of a formal semantics.

In this paper, we propose SOCK, Service Oriented Computing Kernel, which is a process calculus equipped with a formal semantics, for addressing all the basic mechanisms of service communication and composition that takes inspiration from Web Services specifications. Our approach aims at dealing with different service features by considering them separately and in an orthogonal way. We distinguish among service behaviour, service declaration, service engine and services system. The service behaviour deals with the internal behaviour of the service and communication primitives, the service declaration is a description of the service deployment, the service engine is the execution environment where
JOLIE: a Java Orchestration Language Interpreter Engine

Fabrizio Montesi\textsuperscript{1}, Claudio Guidi\textsuperscript{2}, Roberto Lucchi\textsuperscript{2} 
Gianluigi Zavattaro\textsuperscript{2}
On the Interplay Between Fault Handling and Request-Response Service Invocations

Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro

Abstract.
Service Oriented Computing (SOC) allows for the composition of services which communicate using unidirectional notification or bidirectional request-response primitives. Most of the service orchestration languages proposed so far provide also primitives to handle faults and manage the subsequent compensation activities. The interplay between these two aspects is non trivial since, for instance, faults should be notified to the request-response communication partners in order to compensate also the remote activities. We first present a simple orchestration scenario requiring a precise distributed fault handling strategy. We show that this strategy cannot be programmed using current orchestration languages; then, we propose a new style for orchestration programming able to specify the required fault management strategy. Finally, we show the generality of our approach by analyzing its properties and applying it to a nontrivial scenario.

†Research partially funded by EU Integrated Project Sensoria, contract n. 016004.
Dynamic Error Handling in Service Oriented Applications*

Claudio Guidi, Ivan Lanese†, Fabrizio Montesi, Gianluigi Zavattaro
A Framework for Rule-Based Dynamic Adaptation*

Ivan Lanese¹, Antonio Bucchiarone², and Fabrizio Montesi¹

Abstract. We propose a new approach to dynamic adaptation, based on the combination of adaptation hooks provided by the adaptable application specifying where adaptation can happen, and adaptation rules external to the application, specifying when and how adaptation can be performed. We discuss different design choices that have to be considered when using such an approach, and then we propose a possible solution. We describe the solution in details, we apply it to a sample scenario and we implement it on top of the language Jolie.

1 Introduction

Adaptation, evolvability and reconfiguration are hot topics today. Adaptable systems change their behavior, reconfigure their structure and evolve over time reacting to changes in the operating conditions, so to always meet users’ expectations [3]. This is fundamental since those systems live in distributed and mobile devices, such as mobile phones, PDAs, laptops, etc., thus their environment may change frequently. Also, user goals and needs may change dynamically, and systems should adapt accordingly, without intervention from technicians.

To achieve the required degree of flexibility, different research groups have proposed frameworks for programming more adaptable applications [1,13,20,17,23]. For instance, the application code may include constraints on the environment conditions or on the user behavior, and may specify how to change the application logic if those constraints are violated [5]. This approach is called built-in adaptation, and allows to adapt the application if the conditions change in some expected way. However, since the adaptation logic is hard-wired into the application, it is not possible to adapt to unforeseen changes in the operating conditions.

Dynamic adaptation instead aims at adapting the system to unexpected changes [4]. Dynamic adaptation is challenging since information on the update to be performed is not known at application development time. We propose a new approach to dynamic adaptation, based on the separation between the application and the adaptation specification. An adaptable...
The First Language for Microservices
Microservices for fine-grained:
- access policies (avoids some side-effects of multi-tenancy!);
- scalability;
- deployment (multistage continuous integration!).
Global vs Local
Global vs Local | Timing Conditions

- **Faults**
  - Mixed: 4%
  - Message: 64%
  - Atomicity: 32%

- **Triggering**
  - Violation
    - Order: (msg-msg, msg-comp race)
    - 68%
  - Mixed: 4%

---

TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems

saverio.giallorenzo@gmail.com  |  DISI at Unibo  |  Sophia Antipolis  |  Evaluation des Projects
Global vs Local

Cassandra 10 10 10 6
HBase
MapReduce
ZooKeeper

TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems
saverio.giallorenzo@gmail.com | DISI at Unibo | Sophia Antipolis | Evaluation des Projects
Choreographies

```java
line@Admin = getInput( "Insert line to track" );
{
    setLine: Admin( line ) -> DB( line )
  | setLine: Admin( line ) -> BusAgency( line )
};
schdl@BusAgency = getBusSchedule( line );
passSchdl: BusAgency( schdl ) -> Tracker( schdl );
hasNext@Tracker = hasNextStop( schdl );
while( hasNext )@Tracker {
    gps@BusAgency = getPosition( line );
passPosition: BusAgency( gps ) -> Tracker( gps );
delay@Tracker = calculateDelay( schdl, gps );
storeDelay: Tracker( delay ) -> DB( delay );
{
    _@DB = insertDelay( line, delay )
  | hasNext@Tracker = hasNextStop( sched )
}
```
line@Admin = getInput( "Insert line to track" );
{
    setLine: Admin( line ) -&gt; DB( line )
    |
    setLine: Admin( line ) -&gt; BusAgency( line )
};
schdl@BusAgency = getBusSchedule( line );
passSchdl: BusAgency( schdl ) -&gt; Tracker( schdl );
hasNext@Tracker = hasNextStop( schdl );
while( hasNext )@Tracker {
    gps@BusAgency = getPosition( line );
passPosition: BusAgency( gps ) -&gt; Tracker( gps );
delay@Tracker = calculateDelay( schdl, gps );
storeDelay: Tracker( delay ) -&gt; DB( delay );
{
    _@DB = insertDelay( line, delay )
    |
    hasNext@Tracker = hasNextStop( sched )
}
}
Correctness by design and by construction

Choreography

\[(Correct\ by\ design)\]

**EPP**

Endpoint Projection

\[(Correct\ by\ construction)\]

line@Admin = getInput( "Insert line to track" );
{
    setLine: Admin( line ) -> DB( line )
|
    setLine: Admin( line ) -> BusAgency( line )
};

Admin
getInput@UI( "Insert line to track" )
{
    setLine@Database( line )
|
    setLine@BusAgency( line )
};
...

Database
setLine( line );
...

BusAgency
setLine( line );
...
Time for discussion!
Time for discussion!
Dynamic Choreographies*
Safe Runtime Updates of Distributed Applications

Mila Dalla Preda\textsuperscript{1}, Maurizio Gabbrielli\textsuperscript{2}, Saverio Giallorenzo\textsuperscript{2}, Ivan Lanese\textsuperscript{2}, and Jacopo Mauro\textsuperscript{2}

\textsuperscript{1}Department of Computer Science - Univ. of Verona
\textsuperscript{2}Department of Computer Science and Engineering - Univ. of Bologna / INRIA

Abstract. Programming distributed applications free from communication deadlocks and races is complex. Preserving these properties when applications are updated at runtime is even harder.

We present DIOC, a language for programming distributed applications that are free from deadlocks and races by construction. A DIOC program describes a whole distributed application as a unique entity (choreography). DIOC allows the programmer to specify which parts of the application can be updated. At runtime, these parts may be replaced by new DIOC fragments from outside the application. DIOC programs are compiled, generating code for each site, in a lower-level language called DPOC. We formalise both DIOC and DPOC semantics as labelled transition systems and prove the correctness of the compilation as a trace equivalence result. As corollaries, DPOC applications are free from communication deadlocks and races, even in presence of runtime updates.

1 Introduction

Programming distributed applications is an error-prone activity. Participants send and receive messages and, if the application is badly programmed, participants may get stuck waiting for messages that never arrive (communication deadlock), or they may receive messages in an unexpected order, depending on the speed of the other participants and of the network (races).

Recently, language-based approaches have been proposed to tackle the complexity of programming concurrent and distributed applications. Languages such as Rust \cite{rust} or SCOOP \cite{scoop} provide higher-level primitives to program concurrent applications which avoid by construction some of the risks of concurrent programming. Indeed, in these settings most of the work needed to ensure a correct behaviour is done by the language compiler and runtime support. Using these languages requires a conceptual shift from traditional ones, but reduces times and costs of development, testing, and maintenance by avoiding some of the most common programming errors.

This work is partly supported by the MIUR FIRB project FACE (Formal Avenue for Chasing malwarE) RBFR13AJFT and by the Italian MIUR PRIN Project CINA Prot. 2010LHT4KM.
AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications
Technical Report

Mila Dalla Preda¹, Saverio Giallorenzo², Ivan Lanese², Jacopo Mauro², and Maurizio Gabbrielli²

¹Department of Computer Science - Univ. of Verona
²Department 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...
Choreographic Programming is a methodology for the development of executable language (Jolie). Finally, we prove an operational correctness-by-construction approach which, given a global description of a system (a choreography), automatically generates deadlock-free communicating languages, model communications using channel names (e.g., variants of CCS and AIOCJ languages [2, 12]: they both implement a formal model of how communications are concretely implemented. A challenging aspect of how communications are supported at the lower level in a tractable way.

How can we formalise the implementation of communications in endpoint models and implementations can compromise the benefits of choreographic programming: the correctness-by-construction approach that requires the definition of a model with the typical clarifications, including the management of underlying data structures and I/O actions and that EPP preserves this property, thus ensuring correctness by construction then follows from the fact that the syntax (EPP) [8–10, 22, 34]. EPP transforms the global descriptions into endpoint calculi -calculus [23, 24], abstracting from how real-world parallelism, modularity [8, 27], and adaptation [33].

Clearly, a satisfactory answer should preserve the correctness-by-construction methodology of choreographies down to the lower level. EPP provides all the necessary details to formally reason about how communications are supported. Consequently, implementing endpoint calculus [26, 30].

The key contribution of AC lies in its semantics tool [1, 36]. The key contribution of AC lies in its semantics tool [1, 36]. The key contribution of AC lies in its semantics tool [1, 36]. The key contribution of AC lies in its semantics tool [1, 36].
DYNAMIC CHOREOGRAPHIES: THEORY AND IMPLEMENTATION

MILA DALLA PREDA*, MAURIZIO GABBRIELLI†, SAVERIO GIALLORENZO†, IVAN LANESE†, AND JACOPO MAURO‡

Abstract. Programming distributed applications free from communication deadlocks and race conditions is complex. Preserving these properties when applications are updated at runtime is even harder. We present a choreographic approach for programming updatable, distributed applications. We define a choreography language, called Dynamic Interaction-Oriented Choreography (DIOC), that allows the programmer to specify, from a global viewpoint, which parts of the application can be updated. At runtime, these parts may be replaced by new DIOC fragments from outside the application. DIOC programs are compiled, generating code for each participant in a process-level language called Dynamic Process-Oriented Choreographies (DPOC). We prove that DPOC distributed applications generated from DIOC specifications are deadlock free and race free and that these properties hold also after any runtime update. We instantiate the theoretical model above into a programming framework called Adaptable Interaction-Oriented Choreographies in Jolie (AIOCJ) that comprises an integrated development environment, a compiler from an extension of DIOCs to distributed Jolie programs, and a runtime environment to support their execution.

1998 ACM Subject Classification:
• Theory of computation
  Control primitives; Operational semantics;
• Software and its engineering
  Distributed programming languages; Concurrent programming languages;
  Control structures; Frameworks; Formal language definitions;

Key words and phrases: Choreographies, Adaptable Systems, Deadlock Freedom.

Supported by the EU project FP7-644298 HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems, by the GNCS group of INdAM via project Logica, Automi e Giochi per Sistemi Autoadattivi, and by the EU EIT Digital project SMAll.