Runtime verification of service-oriented systems: a well-rounded survey

Reza Babaee and Seyed Morteza Babamir*

Department of Computer Engineering,
University of Kashan,
Kashan, Iran
Email: rbabaeeecar@grad.kashanu.ac.ir
Email: babamir@kashanu.ac.ir
*Corresponding author

Abstract: Dynamic nature of Service-Oriented Systems (SOSs) has made conventional verification techniques such as testing, model checking and theorem proving ineffective. As a result, many studies have been dedicated to verification of SOS. In this survey, we first discuss why the conventional verification techniques are not effectively applicable to SOSs and then stress the necessity of the runtime verification as a dynamic approach that formally checks the system behaviour at runtime. To provide a deep comparison between existing studies and to draw a clear road map for the future studies, we introduce the salient characteristics of the runtime verification in SOSs. In addition, we survey a significant number of various studies in the literature by categorising them in five broad categories: logic and calculus oriented approaches, workflow monitoring, state-based conformity assessment, aspect-oriented verification, and SLA-driven compliance. Then, we deal with the comparison between the defined categories and finally discuss open problems.

Keywords: service-oriented systems; service composition; run-time verification; run-time monitoring; reliability; distributed systems.


Biographical notes: Reza Babaee has received Master and Bachelor degrees in Software Engineering from University of Kashan in 2011 and 2009, respectively. He has several publications in various research areas, and also involved in a few number of large-scale IT consultancy projects supported by National Iranian Oil Production and Distribution Company (NIOPDC) and Ministry of Education in Iran. Since 2012, Reza has been a software consultant in HASEB System Engineering Consultancy Co.

Seyed Morteza Babamir received BS degree in Software Engineering from Ferdowsi University of Meshhad and MS and PhD degrees in Software Engineering from Tarbiat Modares University in 2002 and 2007 respectively. He was a researcher at Iran Aircraft Industries, Tehran, Iran, from 1987 to 1993, Head of Computer Center in University of Kashan, Kashan, Iran, from 1997 to 1999 and Head of Computer Engineering Department in University of Kashan from 2002 to 2005. Since 2007, he is Associate Professor of Department of Computer Engineering in University of Kashan. He authored one book on software testing, four book chapters, 15 journal papers and more.
1 Introduction

With the advent of service computing, a major number of research and studies have been conducted on the fundamental aspects of Service-Oriented Systems (SOSs) (e.g. design, security, testing and verification). The flexibility and compatibility strength of SOSs practically make them a feasible approach for developing open distributed software infrastructures. However, verification of a service-oriented software product imposes significant challenges for their developers. The huge number of end-users exercising different aspects of the SOSs may enforce changes on their business requirements, for instance; this should not lead to the system works incorrectly or unreliably. More importantly, notorious unreliable third-party services that are frequently used in the service-oriented applications, implies that the whole behaviour of the system has to be formally checked against policy models at runtime. Moreover, error and fault diagnosis and localisation in the distributed environment are often tedious and impractical. Therefore a vast majority of studies in the context of service computing is devoted to the verification of SOSs in order to building fault-tolerant and dynamic reconfigurable products.

1.1 Contribution

The significant contribution of this paper is threefold:

1 First, we show that the verification of SOSs can not be adequately obtained by conventional techniques of model checking, formal verification, symbolic execution. Accordingly, a complementary verification process is required to formally check the system behaviour when it is on-the-fly.

2 Second, we introduce the salient characteristics defined for runtime verification of SOSs. They are used for categorising the current studies and for demonstrating the various aspects to be explored for both evaluation of the existing methods and proposition of new techniques.

3 Third, we categorise recent studies in the literature into five categories according to the introduced characteristics. Each category discusses characteristics of one or two major studies called core research work; then, other related studies in each category are explained if there are distinctive characteristics.

1.2 Scope

The major focus of this paper is on the scope of run-time verification of service-oriented systems; however, other essential concepts are explained if necessary. We strive to avoid
filling this paper with specific code, notations, diagrams, or standards. Instead, we propose an overall insight into the subject by providing comprehensive references to the recent research work and developed technologies in the literature.

To the best of our knowledge, there is no comprehensive survey on runtime verification itself. We do not address all research works in the context of runtime verification neither; although, to keep the paper self-contained, we provide a brief introduction on the runtime verification and the other side of the paper, i.e. key SOSs concepts that are widely used in the literature. Interested readers may find thorough details on services computing in Zhang et al. (2007).

1.3 Terminology

Various types of terms are applied to the verification and SOS concepts in the literature. Nevertheless, to keep the survey consistent and more understandable, while briefly list most of them, we proceed with a discussion to select and apply the most appropriate ones throughout the survey.

1.3.1 Verification terms

In addition to verification, another term that widely used in the verification context called analysis. Analysis means careful study or examination of a piece of software (code) to find functional/non-functional properties and diagnosis of faults and errors (adapted from Dictionary (2008) with major modifications). This is why in some technical references Ghezzi et al. (1991), verification branches into analysis and experimental methods. Consequently, static and dynamic analyses are used as synonyms for formal and run-time verification, respectively. However, from definition perspective we believe that the term run-time verification is more descriptive because, as we discuss in Section 2.3, there are not any forms of analysis in some situations and a simple comparison statement is just performed. The verification term is alternatively used to the Validation process, which is an activity in requirements engineering phase for checking the stakeholders’ requirements.

1.3.2 Run-time verification terms

In the current distributed system context, run-time monitoring is frequently used instead of run-time verification. However, a subtle difference is reflected between them. Monitoring means to watch and check the system behaviour carefully for a period of time in order to discover probable faults (adapted from Dictionary, 2008 by minor modifications). It is a requisite stage in the most of engineering activities including software engineering. In the context of software engineering, runtime monitoring generally means to monitor the system execution for verification, system reconfiguration, security purposes or even the human computer interaction issues.

On the other side, the term runtime verification is actually more popular in the software engineering context, i.e. the verification is one of the crucial phase of software development. The distinctive difference between the run-time verification and the run-time monitoring is primarily marked in using formal methods, since the term verification evokes a confidence that is only attainable using formal methods. More specifically, a
reliable system can be formally verified by proving correctness or demonstrating a
counterexample. In addition, run-time monitoring may use informal methods and
accordingly it may to be not able to bring a necessary reliability. In brief, run-time
verification is a specific form of run-time monitoring concentrating on the verification
process. Therefore, we use the term runtime verification to emphasising the formal
specification as an essential element of the run-time verification process.

1.3.3 Service-oriented related terms

Service-Oriented Architecture (SOA) was introduced as an effective enterprise solution
for the heterogeneous environments. This idea was proposed to provide an open software
architecture style in distributed systems. Afterwards, the service-oriented term was used
more often with other terms. Accordingly, the services science and the computing
technology is joint to establish a new discipline called service computing (Zhang et al.,
2007). Some studies apply the term web services or generally use services to discuss
about the service-oriented systems.

1.4 The paper outline

The remainder of the paper is organised as follows. In Section 2, we propose different
types of verification techniques and discuss why runtime verification is just a pragmatic
approach for the reliable verification of SOSs. In Section 3, a primer on the service-
oriented concepts is elucidated.

Prior to proceeding with the current literature, we present general characteristics of
runtime verification of SOSs in Section 4, so that we can critically analyse and
systematically compare studies in the following section. Furthermore, ongoing studies
can include the characteristics have already been unnoticed.

We review the studies that are recently conducted on the runtime verification of SOSs
in Section 5 and they are mutually classified into inclusive categories: (1) Logic and
Calculus Oriented Approaches, (2) Workflow Monitoring, (3) State-based Conformity
Assessment, (4) Aspect-Oriented Verification, (5) SLA-driven Compliance and (6)
miscellaneous ones.

Finally, in Section 6, we discuss the future of the run-time verification of SOSs and
its main purpose, i.e. developing more reliable software products and the post-stages
should be reached, i.e. self-healing and self-adaptation mechanisms.

2 Why is run-time verification of SOSs needed?

Preliminary programs containing a few lines of code (LOC) to advanced ones including
several KLOC programs need the process of checking its functionalities against its user’s
requirements. ‘Verification is the method used to check if a program (software) meets
user’s requirements’ (Ghezzi et al., 1991).

Two basic approaches have been proposed for the software verification: static and
dynamic verification. The former is dedicated to the code inspection in order to find
anomalies by providing simple analysis, formal proofs or counterexamples. As a result,
the static verification process checks properties could be verified without program
execution. The latter concentrates on the software behaviour when it is on-the-fly, i.e. it checks the system behaviour when it is running, during either experimental or actual execution.

2.1 Software testing as a dynamic verification approach

Software testing is the most conventional dynamic verification technique that is employed by programmers. However, it verifies the system behaviour by the test data inputted to the program and then comparing program outputs with the expected values. If they do not match up, the debugging process is initiated and the test process is repeated (known as Regression Testing).

The testing process confronts some formidable challenges when it is applied to SOSs:

- How can we generate test cases so that they imitate actual input data in the service-oriented system? Note that in SOSs, many inputs are provided by external services whose behaviour patterns are not obtainable before running the system.

- How can we take into account unforeseeable effects of the environment where SOSs will be executed? Execution environment comprises of external services, hardware/software platforms, network infrastructures, etc. Simulation seems a straightforward solution, but in the giant SOSs involving a huge number of services, it is practically infeasible.

- How can we deal with the uncertainty of the services behaviour? In other words, some services exhibit entirely different behaviours, when tested. Therefore, supposing all probable input data are exercised in an exhaustive testing, it demonstrates no commitment to the developers in SOSs.

- The degree of confidence that testing gives to developers is not acceptable for ultra-reliable and giant SOSs. In fact, if the possibility of an error occurrence is one per billion, it must be detected before system deployment. No testing technique would help the developers to provide such level of reliability (Goodloe and Pike, 2010). This brings into mind the famous quote that testing shows only the existence of errors not their absence (Ghezzi et al., 1991).

2.2 Formal verification: the dominant static method

For many years, static verification was typically considered as the compilers, debuggers, and other human-centred methods. Challenges in the testing as a verification method (see Section 2.1) direct software engineers to the more advanced and reliable verification techniques. Originally, formal verification was considered to help hardware designers for ensuring the functional correctness of the developed hardware (Drechsler, 2004). However, by the increasing of software complexity in the industrial applications, formal verification becomes applicable to the safety-critical software systems, as well. Formal Verification is a process to prove that some properties exist in the formal specification model (Bjorner, 2006). Thanks to huge static analyser tools, today formal verification is a popular static verification method which is mainly founded on the Formal Methods. Formal Methods are applied to specify the software using mathematical models in order to provide correctness proofs and inconsistencies prevention.
Despite software testing, formal verification automatically guarantees presence or absence of the desired properties within the program. In other words, the mathematical foundation that is provided by formal methods, gives an absolute assurance to the verifier about the checked property.

Nonetheless formal verification suffers from technical shortcomings, especially in the service-oriented climate:

- Formal methods usually specify the system at the model level, and if the realisation of the model to the executable code is not perfect, formal verification will be incomplete and unsound. However, as we discuss in Section 2.3, this problem may appear in a runtime verification approach as well, the impacts are addressed in the process of adaptation.

- Formal verification is performed without executing the program, and all made analyses lack any information about run-time behaviour as well as the operating environment. This causes the verification process to be performed on only the static attributes of the software and more importantly the real effects of run-time environment is overlooked. This problem becomes more serious when applies to the essentially non-deterministic and chaotic environment of SOSs.

- Absence of user-friendly tools, the inconvenience and misunderstanding of working with mathematical notations are additional problems acknowledged for the formal methods, and consequently, the formal verification approach (Woodcock et al., 2009). This causes a dramatic increase in system development cost and time. In practice, except for safety-critical and mission-critical software systems, formal methods are on-the-shelf in other domains.

SOS requires the formality that is brought to by formal verification techniques, although; it also demands a verification approach which can conform to its stochastic and fuzzy environment, eminently when the number of services rises. As we explain in Section 2.3, runtime verification fills the gap between the required formality and the monitoring at the run time.

2.3 Run-time verification: the new dynamic verification method

Despite using conventional verification methods (e.g. software testing), three avionic safety critical systems failed. This promoted further studies for more reliable verification methods, emphatically for distributed systems (Goodloe and Pike, 2010). Run-time verification is proposed to remedy the stated deficiencies in other verification methods (there are also other novel verification approaches introduced recently like model checking, symbolic execution, etc. We do not consider them in this paper since their evaluation, especially in SOSs, is still premature). Run-time verification is a dynamic method, i.e. the verification process takes place by the program execution. In fact, it is a combination of testing and formal verification, to promote more benefits, whilst avoiding the disadvantages in both.

Run-Time Verification is formally defined as follows (Leucker and Schallhart, 2009):

“Run-Time Verification is the discipline of computer science that deals with study, development and application of those verification techniques that allow checking whether a run of a system under scrutiny, satisfies or violates a given correctness property.”
2.3.1 Runtime verification: generic model

In Figure 1, the simple runtime verification model is depicted. The model has two parallel parts. In the first part, there is formal specification model which is supposed to have been already validated. The specification is then realised by the executable program modules. The realisation process is not a critical aspect of runtime verification, but as we see in Section 6.1 the adaptation process might contain technical refinement, reconfiguration, reestablishment, or even re-realisation. The system is deployed, and simultaneously monitored in order to collect events and data for a verification process. Depending on the inclusiveness of the verification process, i.e. the entire system behaviour or a portion of system has to be verified, the mass of data as well as events needed to be logged and monitored are variable. Moreover, the type of verification in a distributed system, determines the layer in which the events have to be monitored, e.g. within the business layer, the application layer or even the physical layer (the abstraction level of monitoring – see Section 4.2.2).

Figure 1  Runtime verification: generic model

In the second part of the model, domain users define the desired properties which should be satisfied by the system during an execution trace. Another crucial realisation process is regulated, in which the formally specified properties have to be checked on the executing components (i.e. services). Therefore, a perfect runtime verification process should provide the most appropriate realisation mechanism according to both specification and development languages (see Section 4.6).

The main power of runtime verification is demonstrated in the conformity procedure within the model. Conformity procedure checks the existence/non-existence of expressed properties in the running system. If it is compliant, the system works soundly. Otherwise, a violation is detected where a procedure should be activated, that in the simplest form raises an alarm or more complexly, autonomously leads the system into a safer state (more in Section 6.1).
2.3.2 Monitors

The main constituent of run-time verification is a *monitor*. The notion of *monitor* is generally defined as follows (Leucker and Schallhart, 2009): “…a monitor is a device that reads a finite trace and yields a certain verdict.” Device can be in-line codes, an autonomous module, a reusable software component (e.g. services), or even a software-hardware co-designed architecture (e.g. Dodd and Ravishankar, 1992). Thus, monitors should retain two empowering features: audition, and determination. From runtime verification perspective, a monitor simply audits the behaviour of a piece of software, and in many situations, it also checks the audited behaviour against the specification model. However, many studies make other devices, other than monitors, responsible for taking decisions (e.g. developers).

Generally, monitors and the target system are executing simultaneously. As a result, they proportionately affect on the overall performance of the target system execution; however, in distributed climate like SOSs, this can be controlled by dedicating exclusive redundant resources to the monitors. More information on the monitors as well as the monitoring process, particularly in SOSs, is explained in Sections 4.1 and 4.2.

2.3.3 Runtime verification privileges in SOSs

Run-Time verification has successfully overcame the main challenges that other verification approaches confront by in SOSs (discussed in Sections 2.1 and 2.2).

- Run-time verification enables us to take into account the dynamic behaviour of the system throughout pervasive runtime monitoring. While run-time verification might seem similar to the testing, because both of them check system behaviour during execution time, their effectiveness significantly diverges in SOSs. In fact, testing has to deal with generating test cases imitating the real inputs, which is often impractical in large SOSs due to their chaotic behaviour. Whilst run-time verification simply accepts the normal input data at the run time.

- Run-time verification copes with the natural execution ecosystem, and therefore, the verification is performed without environment simulation. This is an outstanding characteristic that other verification methods do not retain, and is critically essential for the stochastic and encapsulated execution environment of SOS. However, as we explain in Section 2.4, this characteristic stems from the fact that the verification process is postponed to the run time which restricts the human-intervening activities when deviation from specification is diagnosed.

- Employing formal methods in run-time verification, provides the high degree of reliability that is required in the development process of ultra-reliable SOSs.

- Since the verification process is limited to the current execution path, the problem of program state explosion posing in the model checking verification approach, or formal theorem proofs, is inwardly rectified.

2.4 Run-time verification: post-activities

Run-time verification mainly performs the verification process at the execution time, and consequently, is monitoring the dynamic behaviour of the system, regulating normal
inputs, and coping with natural ecosystem. Nevertheless, the main formidable challenge in run-time verification is arisen when deviation from system specification is diagnosed. Indeed, during the verification process the human-intervening activities for bringing back the system to the normal state are relatively restricted (unlike static methods and testing which have more flexibility in recovering the system from failure states by the human-intervening methods). Hence, proposed run-time verification techniques are typically followed by post-activities which automate (with least human-intervention) the recovery process of the system.

The simplest strategy is to send a warning message to the user, demonstrating that some deviation from system objectives is detected (or anticipating that deviation will be dissociated). Afterwards, the consequent corrective actions are taken by the operator (Leucker and Schallhart, 2009), which is the most intervening form, and basically similar to other verification approaches. For instance, in Schroeder (1995) monitor only logs events which are causing an error so that the records are analysed and then the recovery process is derived. This approach is basically an off-line monitoring process (described in Section 4.2). In Dodd and Ravishankar (1992), the monitor is used like a code debugger and according to the found errors, displays appropriate messages.

However, there are also extensive studies on mechanising corrective actions to help the system recovers from diagnosed strains. Kim et al. (2002) define a constructive role for monitor within the system that when a violation is discovered, monitor tries to take the system back to the previous safe state, like an undoing operation.

The trend of self-healing systems, in which the system automatically fixes the revealed flaws by runtime verification process, continues with general concepts like self-adaptive, or evolvable software systems (Barringer et al., 2007). Self-adaptive or evolvable systems, supported by monitors, attempt to adapt themselves to not only uncertain deviation from system specification, but also to the new user requirements and situations. Detailed studies on self-adaptive software engineering are discussed in (Cheng et al., 2009); although, there is a long way to reach a novel architecture for these systems (Brun et al., 2009).

3 Service-oriented system primer

From the beginning era of distributed computing systems, the heterogeneity of interoperability between systems were seemingly intractable. The problem is gradually revealed in different layers within a distributed system (e.g. physical layer, application layer, communication layer, business layer, etc.).

Service Oriented Architecture (SOA) is an architectural style to provide an open, flexible, and integrated interoperability layer between distributed IT applications (McGovern et al., 2006). The key enabling aspect of SOA is to reuse existing services from multiple providers without major changes in underlying technology, to create new business processes and satisfy customer requirements (Zhang et al., 2007). Shortly after exposition of SOA, other service-oriented concepts emerged into other research topics: Service-Oriented Real-Time Systems, Service-Oriented Requirements Engineering, Service-Oriented Operating Systems, Service-Oriented Cloud Computing etc., and that is why the reviewed research work use various terms while framing the same concept. To make the paper more consistent, all these terms are conceptualised in a single notion, we call Service-Oriented System (SOS).
In an SOS, all services descriptions, interactions, compositions, discoveries, and even executions are based on XML open standard format. Formally, SOS is defined as follows (Zhang et al., 2007):

“Services System can be viewed as a self-contained encapsulated system providing some services to the outside world.”

In comparison to an object-orientated philosophy reflecting that everything is an object, SOS is in the doctrine of everything is a service. In other words, service-orientation is the system development process based on the entities called Services: “self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications” (Papazoglou, 2003). By definition, SOS is a closed-loop feedback system, i.e. it continuously reacts to the execution environment, and consequently, should be rigorously monitored according to the environmental dictates at the run time. As a result, runtime verification process is a constitutive element of SOS perpetuation.

SOS is facilitated by the advent of IT-enabled services (e.g. web services) and the need to align business services with new IT technologies (Zhang et al., 2007). Web services indeed attempt to tackle the problem of interoperability within the distributed systems. Zhao et al. define Web Services as a new industrial standard for distributed systems that provide universal interoperability for each system (Zhao et al., 2007). Nevertheless, permeating web services into the implementation of distributed applications, poses more significant interoperability problems among businesses. In other words, many enterprise applications need to cooperate with third-party web services to execute a complicated business process (business level inter-operation). Leveraging SOSs hence, enables distributed applications to achieve an objective standard for web services inter-communication, integration and composition in order to realise a business process.

3.1 Service oriented architecture

**Service-Oriented Architecture** (SOA) mainly has three fundamental parts (Figure 2): Service Provider, Service Consumer, and Service Repository. The service consumer (which is referred as service requester in some contexts) requests for a desired service which might be discovered within the service repository (which is also referred as service broker in some contexts) based on a set of varied filters. It then receives binding information for invoking the selected service from the service repository. Each service provider should fully register new services on the service repository with the binding information. These are logical roles, i.e. a physical system (or service) may play the role of a service provider, and a service consumer, concurrently.

*Figure 2* The simple SOA model
A vivid description of web services must thus be published in the service repository so as to enable an advanced search by the service consumer. The description is usually expressed in the Web Service Description Language (WSDL). The format of messages exchanged between services is proposed in the Simple Object Access Protocol (SOAP), which is an XML-based document, supporting all kinds of services interactions.

REpresentational State Transfer, or REST (Fielding and Taylor, 2002), was introduced by Fielding to establish a modern architectural style for developing hypermedia web applications. In fact, SOA may rely on REST, instead of SOAP, as an architectural style which imposes constraints on the data elements, connectors, and the processing components (Fielding and Taylor, 2002); although, SOAP and REST are diverging in their viewpoints. SOAP along with WSDL is a framework protocol approach that has no global addressing scheme, supporting RPC-like approach, with more flexibility and extendibility, while REST uses HTTP standard methods for inter-operation between web services and supports URI as the main global addressing scheme (Jakl, 2006). It is claimed that in Amazon web service provider, REST interface is more popular than SOAP interface; however, we could not find any studies in the runtime verification of SOSs that account REST. For more discussion on the comparison of SOAP and REST, interested readers may refer to Przybilski (2006).

3.2 Service discovery

The expected result of SOS is to reuse existing services and deliver new ones by combining them. However, prior to the service composition, SOS should support the customers in discovering appropriate services based on their business objectives. Hence, SOS requires a repository to store/retrieve services information and modify the existing ones, which is called a Service Repository.

A service repository might be deployed in the two following ways: centralised and decentralised. A centralised registry maintains the links for the published services, information about binding, previous interactions, the providers etc. that can be simply searched by user to discover the most proper services for creating new composite services. Universal Description, Discovery and Integration (UDDI) is a centralised service repository in which the services can be discovered by providing searching criteria. When search criterions are matched with a published service entry in UDDI, the links and other binding information are sent back to the service requester.

Decentralised deployment relies on every service provider in the system which preserves a simple document providing the descriptions of the published services. The document also supports for aggregation of referencing documents from other service providers to cover all published services. Once a service consumer requests for desired services in the nearest published document, the corresponding service provider inspects the document for a match between the published services and the searched criteria. If a complete match is found then the service provider will return the binding information, otherwise it will return the referencing documents of other service providers. The search operation in the chained documents proceeds in turn until an appropriate published service is discovered. Web Service Inspection Language (WS-Inspection or WSIL) is a typical XML-based service registry document supporting decentralised service discovery.

In both of service discovery deployments, the search results may contain several services that satisfy users criteria. In this situation, an optimisation auxiliary process can be driven to select the most perfect service according to the added value and costs,
quality of services (see Section 3.4) and the composition of services (see Section 3.3). The optimisation process is not in the scope of this paper; interested readers may refer to Zhang et al. (2007) for more details.

3.3 Services composition: realising a business process

The main goal of SOS is to create an integrated and flexible interoperability layer between several enterprises with various services standards. Enterprise Application Integration (EAI), e.g. CORBA, DCOM (Chung et al., 1998), etc., provides an integrated layer between services of a single enterprise, but the EAI platforms are usually incapable of fostering interoperation between services from multiple providers (McGovern et al., 2006). In practice, meeting the business-level requirements needs not only intensive internal services combination but also harshly demands collaboration with third-party provided services. For example, a travel agency system requires internal or external services from airline companies, hotel reservation systems and car rental services in order to provide a consolidated travel management process. In brief, services provided from multiple independent sources should be composed to reify business process models within an IT-enabled climate. Services are composed by the Orchestration and Choreography mechanisms.

3.3.1 Orchestration

Orchestration refers to a centralised agent which is responsible for composing services, i.e. all web services interactions are coordinated by a centralised coordinator to form the specified business process. In other words, the business logic, that is implemented by the services composition, belongs to a single party that may not even provide services. In fact, the aggregator agent offers a new value added composite service by combining existing services. To define the sequence of services combination which implements a business process, the aggregator may use Business Process Execution Language (BPEL) that specifies a proper chain of services execution using structured activities like sequence, flow, and pick.

3.3.2 Choreography

Despite orchestration, choreography pursues a more decentralised approach: each service has to handle its own interactions so that the specified business process is obtained. In a choreographed SOS, all involving parties may publish the composite service as a new provided service, i.e. the logic of the implemented business process belongs to all involving parties. Choreography needs a particular language, like Web Service Choreography Interface (WSCI) (see http://www.w3.org/TR/wsci/), to define the services conversation with other participating services. There should also be an advanced discovery framework, like DSDF (Zhang et al., 2004) that overwhelmingly supports the service discovery in the service-oriented ecosystem.

3.4 Quality of Services (QoS)

In an intensive competitive service oriented ecosystem (e.g. Cloud, or P2P SOSs), in addition to the functional requirements, the quality of provided services is clearly crucial.
There should be also precise mechanisms that allow consumers to monitor negotiated services behaviour and quality, to verify whether they are compliant with user’s requirements. Service Level Agreements (SLA), Operational Level Agreements (OLA), and Business Level Agreement (BLA) are formal descriptions used to demonstrate contractual agreements between service provider and consumer in different levels of service-oriented hierarchy. They describe the non-functional properties, or Quality of a service (QoS) that are negotiated and have to be satisfied by the provider.

The most widely used metrics for QoS and their measurements in SOSs, adapted from Ghezzi and Guinea (2007), are as follows:

1. **Dependability**: which is defined in the sense of three aspects:
   - **Availability**: which is a measurement of how long the service is available and ready to operate.
   - **Reliability**: which measures the possibility of the service failure and the time it needs to recover.
   - **Accessibility**: which measures the time when the service is accessible even in heavy load circumstances, and also specifies the potential scalability of the service.

2. **Security**: which defines the authorised accesses to the service.

3. **Performance**: which is itself defined in terms of two aspects:
   - **Throughput**: The number of requests (or services) that the service provider can respond to in a timely manner.
   - **Latency**: The back-and-forth travel time of the service request (round-trip time).

Note that the runtime verification framework can be built to monitor both the functional and non-functional behaviour of a service. To verify the functional requirements, a formal specification and consequently formal method is necessary (see Sections 2.3.1 and 4.3), but the verification of non-functional requirements can be aligned with any formal agreements and metrics at different levels (e.g. SLA, OLA, or BLA) between the consumer and the provider. We discuss verification of both functional and non-functional requirements in Section 4.7.

### 4 General characteristics

To draw a meaningful comparison between the current research work, first we provide the salient characteristics for the runtime verification framework in SOSs. Some of the presented features are adopted from Ghezzi and Guinea (2007), although, we strive to propose more detailed explanation and dimensions on the general characteristics. Note that some of these characteristics are inextricably intertwined with each other that altering one would affect the others, and so we also describe the mutual dependency of various characteristics.
4.1 Monitors

As discussed in Section 2.3.2, monitors are the central core of runtime verification process, and when devising runtime verification in SOSs, several dimensions of monitors should be contemplated: the level of monitor intrusion into the subject programs, and the distribution degree of instantiations of monitors.

4.1.1 Level of intrusiveness

Monitors are generally classified by the level of intrusiveness into two types: intrusive and non-intrusive monitors. Intrusive, or in-line monitor, is whose code is inserted into the subject program, either manually or automatically. Non-intrusive, or out-line monitor, is an autonomous program that might even need its exclusive resources (computation, memory, network bandwidth, etc.). In other words, an intrusive monitor shares its resources with the monitored program, whilst a non-intrusive monitor is usually allocated exclusive redundant resources.

Either intrusiveness type has its own advantages and disadvantages, that make monitors fairly competent in various ambient conditions. Intrusive monitor is applicable in every application whose code is readily available and modifiable so that the necessary parts of the subject program source code can be simply altered. Intrusive monitors often employ instrumentation techniques to justify assertion instructions in the program, perform exception handling, or check the state of the program. Aspect-Oriented Programming (AOP) is a programming technology that enables implementing intrusive monitors to define aspects of the subject program that should be monitored by defining join points, and automatically weaving the monitor source code into the program as pointcuts. We describe AOP approaches in Section 5.4.

On the other side, non-intrusive monitor is particularly applicable in distributed environment where the monitor and the subject program are likely executed at the same time. Respecting the subject program that counts performance costs of the intrusively monitoring process, non-intrusive monitor is less prohibitive if more redundant resources are provided. Indeed non-intrusive monitors are still prohibitively costly in the time shared systems. From pragmatic point of view, requesting third-party provided services is practically widespread, specifically when composing services to create business processes, therefore, supposing that the source code of the software as a service, is available or at least modifiable for intrusive monitors, are often quite impracticable. Furthermore from definition point of view, services are encapsulated black-box entities that only provide necessary interfaces for invocation (Section 3), and thus, intrusive monitor violates the platform-agnostic principle of SOS. Nevertheless, in some studies (Ghezzi and Guinea, 2007; Baresi et al., 2004; Huang et al., 2009) intrusive monitors are employed to alter service-oriented execution frameworks (e.g. BPEL) which present a modest platform-agnostic verification process.

4.1.2 Degree of distribution

Instantiations of monitors in a distributed system may be deployed in two broad ways. The first more predominant one, is Centralised deployment (e.g. Arshad, 2006) in which the instances are placed in a single system to entirely monitor every role in SOSs, and diagnose any failures according to the developed specification. In Decentralised
deployment (e.g. Bauer et al., 2006), the monitor instances are deployed on the individual systems that communicates to each other to find possible violations throughout the entire SOS. If the number of monitor instances is equal to the system roles, namely that each service provider, consumer, or broker, has one instance of the monitors, the deployment is called Widely Distributed Monitors (Sen et al., 2004).

Both in centralised and decentralised deployments, there are extreme subtleties that should be appreciated in SOS; however, some of them stems from the nature of distributed system itself: (1) causality order of the exchanged SOAP messages, particularly in the widely distributed deployment, (2) the volume of traffic generated by the multiple monitor instances especially in the decentralised deployment that might overcook both monitors and subject entities (e.g. service provider), (3) Providing updated information on the service entities for all monitor instances (e.g. UDDI issues any updates on the published services). The structured approaches should investigate these issues regarding the level of distribution of the monitor instantiations.

### 4.2 Monitoring process

Whatever monitor instantiations deployments are planned, the monitoring process itself may be dissent in two dimensions: the time when the monitoring process is performed, and the abstract level of monitoring.

#### Figure 3 Monitoring timeliness spectrum

4.2.1 **On/offline modes**

Respecting the time of monitoring process, it can be driven in either an online or offline mode. Online monitoring process is identical to the event-based monitor invocation where monitors are activated by the event occurrences. Offline monitoring process is similar to the time-based monitor invocation where monitors are activated in the certain time intervals. In an off-line mode, although the required data for monitoring is collected when the system is on-the-fly; the logged events are scrutinised when the system execution passes the time of logged events (due to termination, suspension, or continuing execution). As a verification method, the off-line mode is applicable to the systems that are not obligated to be verified in a timely manner (e.g. most of information SOSs), and instead may be taken as a promising approach for only monitoring purposes. Process Mining (van der Aalst and Weijters, 2004) is an off-line monitoring technique that extracts relevant information from event logs which can be also used for verification purposes which is called conformance checking (Aalst et al., 2008), i.e. checks the conformance of a workflow with the customer objectives (see http://www.processmining.org/ for more details).
In an on-line mode, both data acquisition and analysis are conducted in a timely manner when the system is on-the-fly. On-line monitoring mode is relatively much closer to the concept of runtime verification (see Section 2), as the whole process of monitoring is performed during execution time.

Note that both on-line and off-line modes are indeed at the two ends of the spectrum of timeliness of monitoring process (Figure 3). On-line monitors may be deployed for real-time verification of SOS safety properties, whilst employing off-line monitors could be only for the purpose of performance measurements. As shown in Figure 3, in many SOSs, the monitor services are settled in the middle of monitoring timeliness: they usually monitor the behaviour of the services composition in certain periods of time. Nonetheless, as we see in Section 5, most of the research work incline towards on-line monitoring process which is more closer to the runtime verification established goals (see Section 2.3.3).

4.2.2 Level of abstraction

The monitoring process may involve different levels of abstraction. Discussed abstraction levels cover both data abstraction and procedural abstraction. However, the data abstraction level is more critical for a runtime verification process:

1. Business-Level: The highest level of monitoring, in which the verification process is aimed at the business logic, and therefore, it has to capture the data that represents the logical sequence of the business process execution. The business-level monitoring should be provided with high-level requirements that reflect multiple business processes objectives.

2. Process-Level: The second highest level of monitoring, is the process-level that conventionally monitors a single business process. Therefore, the monitoring data reflect the sequence of services invocation (see Section 3.3).

3. Message-Level: The sequence of exchanged messages between SOS roles, could be scrutinised to verify its behaviour. Since service invocation procedure are indeed delivered via passing messages, the message-level monitoring is basically identical to the process-level monitoring. Nevertheless, if the logical objective of the verification can be interpreted by a combination of exchanged messages which do not reflect any business logic unless the service interactions, the message-level monitoring can be regulated, e.g. in Gan et al. (2007).

4. Service-Level: Assuming Software as Service, the monitoring process can be unfolded in a single service, representing the software that presents the service. Hence, the service-level monitoring is simply the verification of the developed software which acts as conventional software verification processes requiring only local data and control flow. Similar to the process-level monitoring, the verification of software as service can be achieved by auditing the status of participating services, irrespective to their interactions (Paul and Tsai, 2004). Notice that as the correctness of implemented interfaces within the software as service, obviously impacts the way that it interacts with other services, and generally behaves in SOS ecosystem, service-level monitoring is absolutely necessary. Moreover, the QoS-related deviations in the engaging services that may affect the overall execution of a
composite service, clearly demonstrates the service-level monitoring as a crucial step in the verification of particularly Grid or Cloud Computing SOSs (Section 3.4). On the other hand, monitoring encapsulated self-contained services in SOS operating environment, might not be technically feasible. The solution is covered by the concept of service certification (Scholz et al., 2005), that provides the service requestor with ensured enabling documents, eliminating the performance barrier caused by service-level monitoring.

5 Low-Level Events: At the lowest level, the monitoring process involves in lowest generated events in the system. In SOSs, all the low-level events are expressed by SOAP massages (or other application-layer messages, e.g. HTTP messages) which is the standard XML-based format for messaging between service-oriented entities (Section 3.1). Any system behaviour verification which refers to only one single low-level message (e.g. SOAP), is driven by low-level events monitoring.

4.3 Formal specification language

Formal specification languages play a prominent role in runtime verification process (Section 2.3). Therefore, determining a theoretically well-founded, and technically practical formal specification method respecting to the monitors deployment, monitoring process, development language, the level of ensured reliability by the verification process and also the domain requirements restrictions, is a key enabling aspect of runtime verification, particularly in SOSs.

The formal specification language can be applied in two ways in the runtime verification process (see Figure 1): (1) to specify the system functional behaviour and requirements, and (2) to specify the desired properties and constraints that should be satisfied by the system implementation. In many approaches, the desired properties and constraints are distilled from already developed system specification (Robinson, 2003). Nonetheless, drawing comprehensive system specification particularly in large SOS operating environment where a vast number of third-party provided services does not have a complete formal specification, is not always technically feasible. Therefore, many studies in the literature propose the formal specification of the desired properties and constraints for verification, which may have been extracted from either the informal domain requirements or even the system development (through reverse engineering).

Selecting an appropriate formal specification language among the huge number of invented formal modelling and specifying methods, can be made by various policies. Some research work is adhered to the SOS-based standard languages (Annotated BPEL Baresi et al. (2004), Web Service Constraint Language (WCoL) Baresi and Guinea (2005b), Extended Web Service Policy (E-WS-Policy) Narendra et al. (2007), Interaction Property Pattern (IPP) Li et al. (2005), Nested XPath Expression in WS-Policy Charfi et al. (2007) etc.) which have to be formalised to eliminate any ambiguity in the specification. The remainder of research sticks to the prescriptive formal methods like the temporal logic family (Event Calculus (EC) Spanoudakis and Mahbub (2006)), goal-oriented specification languages (Knowledge Acquisision in Automated Specification (KAOS) Robinson (2003)), semi-formal visual modelling languages which is transformed to a formal format (UML 2.0 Sequence Diagrams (SD) Simmonds et al. (2008)), or even more implementation-driven languages, like AOP (History-based Point-
cut Language Huang et al. (2009)). Some research studies successfully combine both prescriptive formal methods with the SOS-based standard languages to propose a hybrid approach (ECAssertion Mahbub and Spanoudakis (2007) and CLIX Baresi et al. (2004)).

We proceed to examine employed formal specification languages in the context, from the following perspectives: degree of expressiveness, and the degree of automation they support the developers in the derivation of desired properties and constraints.

### 4.3.1 Degree of expressiveness

The expressiveness power of the formal specification language must suffice to rigorously express the distributive properties and constraints of the SOS. In addition, since the customer-satisfied properties and constraints of the SOSs are usually defined in the high-level of abstraction, the formal specification language should also provide the capability of expressing abstract specifications. Furthermore, to refine the high-level abstracted objectives into more concrete constraints, the formal specification language has to provide an accurate refinement method so that the monitors can examine execution trace against the concrete constraints. For instance, a business-level objective may force the SOS to only discover the services that produce value-added services in the business process. This high-level requirement should be specified by abstract semantics in the formal specification language (that may be further converted to more concrete requirements by the refinement methods). The level of abstraction of the monitoring process (Section 4.2.2), and the mass of required data for verification, is usually dictated by the formal specification language.

### 4.3.2 Degree of automation

As denoted in Section 2.2, the difficulty of working with mathematical notations, reduces the pragmatic acceptance of formal methods among industry engineers. In practice, monitor construction is technically deliverable by incrementally analysing system formal specification (Geilen, 2001), or even the system implementation through reverse engineering. In huge SOSs however, manually constructing the monitors is relatively cumbersome, and hence, enforces the need for mechanising the monitoring extraction from system specification or implementation. As a result, one of the practically enabling aspect of the formal specification language is the capability of automation it supplies to extract the monitors. For example in (Robinson, 2003) web service monitors are automatically obtained from formally specified requirements using a Goal-Oriented approach.

The ultimate level of automation is when the requirements engineer is facilitated by the analytical tools which can directly acquire the formally specified system functional/non-functional properties, from the informal/semi-formal specification models. From requirements engineering viewpoint although, bridging the gap between requirements gathering techniques and formal specification, is one of the potential favourite research directions in the future (Nuseibeh and Easterbrook, 2000).

### 4.4 Development language

To dynamically verify a system, the system should be developed and executed in the operating environment to fully monitor unexpected situations and verify the system
behaviour (see Section 2). Consequently, the system development language has a
discernible impact on the runtime verification in the SOSs. Despite formal specification
language (Section 4.3), the development language selection is restricted by SOS-based
languages like BPEL. Therefore the number of development languages employed in
SOSs is basically limited. Nevertheless, many research studies focus on the non-
development language aspects to verify the system at the runtime. For example, in
Li et al. (2006) the verification process is led by monitoring the service interactions and
exchanged messages, ruling out the need for considering the development language.

Notice that the underlying implementation languages used to develop services is not
accounted, simply because services as software are encapsulated entities that should
provide verification and certification per se.

4.5 Monitoring data

The monitoring process is driven by data obtained from the subject system and the
operating environment at the runtime. In SOSs (and more widely in distributed systems),
the monitored data is likely to be captured in the form of events, and thus, an event
listener is regularly employed in the runtime verification framework (Gan et al., 2007;
Spanoudakis and Mahbub, 2006). The captured data is mutually dependent on the level
of abstraction of the monitoring process (Section 4.2.2). Therefore, for each level the data
has to be captured and the consequent events have to be listened, considerably differ.
However, the high level of abstraction of data and events may be converted to lower ones
as the refinement process proceeds.

4.6 Realisation mechanism: from declarative specifications to executables

The key idea behind runtime verification process is revealed in the realising mechanism
that translates the desired properties and constraints to the executable programs. In other
words, the significant challenge is how to check the (in)correctness of formally specified
properties and constraints in the executing programs. This complex mechanism is
regulated by the monitor implementation. The monitor is indeed responsible for checking
the (in)correctness of the desired properties and constraints in every trace of the system
execution.

The main contribution of the most research work in the literature thus, is made in the
specification-to-execution transformation mechanism. For instance, Robinson (2003)
exploits two different-time models: design-time model and run-time model. In the
design-time model, the monitors are extracted from goal-oriented requirements
expression by performing the obstacles analysis (van Lamsweerde and Letier, 2000). At
the runtime model, including system design and implementation an ontology map
between the obstacle models and the run-time web services are defined which actually
acts as the realisation mechanism.

Simmonds et al. (2008) contrived an alternative approach, in which both desired and
undesired properties and constraints are specified by UML 2.0 Sequence Diagrams
representing safety and liveness requirements for a system, respectively. The
specification diagrams are then translated into a Definite Finite Automata (DFA) whose
states will be updated according to the web service conversation. The update process is
taken place by the monitors which also check the validity of the current state according to
the safety and liveness requirements. In fact, the realisation mechanism of formally specified properties and constraints to the executable entities, are fulfilled by gradually transforming both into an intermediate model like state machines.

4.7 Desired properties and constraints

The properties and constraints are actually the requirements and concerns that the stakeholders follow in the finalised implementation of the system. There are mainly two types of properties and constraints: functional and non-functional. Functional requirements are simply what the system should do. In service-oriented ecosystem, functional requirements are the assigned task to each service that should be accomplished by the associated service provider during the determined period. Non-Functional concerns, or Quality of Services (Section 3.4), describe how the operational aspects of the system should be performed. Non-functional concerns generally refer to the dimensions which relate to the whole of the system (total view), while each functional requirement refers to the associated part of the system that is determined by the stakeholder or developer (partial view).

To a certain degree, the type of the desired properties and constraints dictates the formal specification and development languages (Sections 4.3 and 4.4), or vice versa. Given the execution description of the entire system (e.g. BPEL), the functional requirements can be captured by analysing system documents which reflect the functionality of several parts of the system. For instance, Spanoudakis and Mahbub (2006) extracts functional requirements from BPEL documents by identifying primitive events that have to be monitored. Conversely, non-functional concerns normally require distinct specification language (e.g. WS-Policy). For example, Tsai et al. (2004) extends WSDL to enforce the non-functional system requirements such as performance, security and reliability, in addition to the functional scenario. Narendra et al. (2007) presents an AOP approach to monitor the non-functional properties specified in an extended WS-Policy document by Nonfunctional-Property tag elements.

The target properties and constraints sometimes affect the level of abstraction of the monitoring process, as well. In many cases, verifying that non-functional requirements (e.g. monitoring security issues) are fulfilled by the SOS, requires monitoring process at the refined abstraction level, such as message-level (Ghezzi and Guinea, 2007).

4.8 Service composition

As the predominant feature of SOSs is the service composition to produce new services (see Section 3.3), the process of runtime verification has to envisage the paradigm of service composition so as to both monitor SOSs and diagnose violations, systematically.

Orchestrated services can be a single point of failure (SPOF) because service combination is handled by a single entity, i.e. the orchestrator. In verification view, in order to verify the SOS behaviour, the orchestrator has to be monitored. However, the monitoring process becomes simplified when we use a centralised event organiser and leave monitoring other entities (e.g. service provider).

Choreography takes advantages of the distribution approach that enables monitoring process relatively dependable. Nonetheless, the efficiency of this paradigm slightly decreases because of the volume of overloaded messages that is exchanged for the purpose of verification.
The distribution level of monitors (Section 4.1.2) and monitoring data (Section 4.5) is highly intertwined with the paradigm of service composition.

5 Research work

In this section, we succinctly review undertaken studies on runtime verification of SOSs. Carried out studies are considerably diverse such that some of them focus on the service-orientated aspects and just pave the way with a monitoring framework, whereas other studies highlight the verification aspects and propose new formal specification languages or adapting the current ones to the service-oriented ecosystem, for instance. Therefore, for the sake of clarity we present a detailed description of representative samples of research work that are classified into several categories. Each category holds one or more of the general characteristics described in Section 4. Nevertheless, note that the categorisation is not strictly accurate and mutually exclusive, and a research work is included within a category according to the hallmarks of the study. General categories and the embodied key characteristics are as follows:

- **Logic and Calculus Oriented Approaches**: Studies which introduce well-founded formal specification language for verifying SOSs using formal logic (e.g. LTL) and calculus (e.g. EC).
- **Workflow Monitoring**: Studies which focus on the governance of services composition, and checking it against pre- post-conditions.
- **State-based Conformity Assessment**: Studies which apply state-based modelling of specification and execution trace, and then assessing their conformity.
- **Aspect-Oriented Verification**: Studies having close relationship with development languages that verifies the crosscutting concerns throughout the SOS.
- **SLA-Driven Compliance**: Studies which check system behaviour against formally specified soft goals that are defined in the contracts.
- **Others**: Other major individual studies.

Every research work category is introduced in four subsections: the first subsection, called Verification Framework, describes the principal components and their relations that form the basic constituents of runtime verification process. The second subsection, called Characteristics, identifies individual characteristics of the work based on the general characteristics explained in Section 4. The Discussion subsection, discusses advantages and disadvantages of the introduced work in each category, and finally, retrospective relationships of the core work in each category with other relevant studies are explored in the forth subsection, named as Related Work.

5.1 Logic and calculus oriented approaches

Logic and calculus oriented languages receive many attention in the community of formal methods due to the theoretical strength in accurately specifying the desired properties and constraints. The core study in this category is the research work of
Mahbub and Spanoudakis (2007), in which they develop the verification framework to monitor the service level agreements using event calculus that is particularly competent in specifying temporal properties.

5.1.1 Verification framework

The verification framework (Figure 4) has several parts:

1. **Monitoring Manager**: supports initiating, coordinating and reporting outputs of the monitor instances.

2. **Event Receiver**: is responsible for gathering the relevant events from specific service composition executor, and records them into the event database.

3. **Monitor**: an instantiation of a monitor that retrieves already recorded events in the database and accordingly, checks the system behaviour against the desired properties, then records a deviation into the deviation database if the requirements are violated.

4. **Behavioural Properties Extractor**: extracts the behavioural (functional) properties of the system from BPEL document and translates them into a specific version of event calculus, called EC-Assertion.

There may be also other auxiliary components in the verification framework for utility purposes, such as user interface, simulator, or specific database handlers. The execution process, which is provided by producing the service composition events, is assumed to be totally separated from the verification process.

**Figure 4** The Mahbub and Spanoudakis (2007)’s simplified verification framework
5.1.2 Characteristics

The principal characteristics of the category are as follows:

- **Monitors**: Decoupling the operating environment and the verification framework implies non-intrusive monitor employment; however, the distribution level of monitor instances is undefined, i.e. there is no explicit implementation evidence demonstrating deployment of distributed monitor instances.

- **Monitoring Process**: The existence of the deviation database in the verification framework indicates that the monitoring process is not real-time; although, it may still remain on-line. Moreover, since the database records are retrieved by the monitor instances chronologically, the rate of event occurrence is slower than the monitoring process pace, and thus, the timeliness of the monitoring process shows a trend towards off-line monitoring. However, using the push mode as the monitoring report mode, in which the violations are reported automatically to the predefined destination at modifiable regular time intervals, user can manually determine the timeliness of the monitoring process.

  The level of the abstraction of the monitoring process is configurable by extending the WS-Agreement (Andrieux et al., 2005) specification to include policies for both service-level and the process-level monitoring. However, the main contribution of the sample research is valued for the verification of service composition (i.e. process-level) against functional requirements (extracted from BPEL or WSDL documents), and the QoS (or non-functional) properties, both described in the WS-Agreement documents. In addition, the WS-Agreement specification is extended to include meta-data for monitoring, and consequently, provide partially reconfigurable monitoring process (Mahbub and Spanoudakis, 2007).

- **Formal Specification Language**: The dominant characteristic of the work category is exhibited by the theoretically well-founded formal methods that are used to formally draw the desired properties and constraints. Mahbub and Spanoudakis (2007) adds extensions to the WS-Agreement documents so that firstly addresses the lack of policies specification for the monitoring process, and secondly to insert both functional and non-functional properties of interest into the document. The latter is specified using an XML schema based on Event Calculus (EC) (Shanahan, 1999), called EC-Assertion. EC-Assertion is identical to EC other than it employs specific events to describe the service guarantee terms, assumptions and conditions. Therefore, the temporal expressiveness of EC-Assertion is the distinguishing feature of the proposed approach. EC-Assertion is a hybrid method that applies both first-order logic and XML standard schema to form a SOS consistent format.

As shown in Figure 4, there is a behaviour properties extractor within the verification framework that processes BPEL document and generates the specification of behavioural properties in event calculus. Hence, the sample research work provides slight degree of automation in extracting functional properties from development BPEL documents (Mahbub and Spanoudakis, 2004a). Nonetheless, non-functional constraints are provided by the user in an envisaged user interface.

Hallé and Villemaire (2009) employ Linear Temporal Logic (LTL) to specify system properties that is further translatable into the X-Query expressions (Hallé and
Villemaire, 2008). There are also two extensions for LTL to define data correlations and time constraints in the business in order to monitor the choreographic constraints.

- **Development Language:** The main widely used development language is BPEL in the research category. All web services participating in the BPEL (business process) should have the WSDL file indicating their individual description. The BPEL document plus containing WSDL files are provided in the processSpecification element of complex XML type MonitoringPolicyType in the extended WS-Agreement contract (meta-data for monitoring). It is noteworthy that the experimental aspect of the core work in the research category includes a Java implementation in which the built prototype is executed by the bpe14j (this technology has been retired by IBM. See http://alphaworks.ibm.com/tech/bpws4j). BPEL execution engine and the corresponding generated events are logged in the log4j format for the monitor instances (see http://logging.apache.org/log4j/docs/ for more details).

- **Monitoring data:** Like many other studies, the subject system in the research category is an event-based system, and the monitoring data are acquired from generated events by the execution engine. Therefore, events type is directly determined by the generator, i.e. the execution engine, that is an external entity in the verification framework (it belongs to the operating environment). However, as the monitoring process is driven at the process-level, the monitored events are usually recorded as the service execution time that represents the reification of service composition.

The other major contribution of the core research is to diagnose the possible violations of expected system behaviour. For this reason, there are two types of events that are captured during runtime monitoring: recorded events and derived events. The recorded event is the primary event captured during system execution, and the derived event can be deduced from current recorded events. The derived event enables the verification process to find possible inconsistencies arisen from system expected behaviour, that is not explicitly stated by the system specification.

- **Realisation Mechanism:** The realisation mechanism is designated as the comparison of generated events (execution program) with constraints asserted into EC-Assertion document (formal specification). In fact, the recorded events are transformed into the EC form by defining them as fluent initiation or termination. The recorded behaviour is then compared with the service guarantee terms, which have been already specified in an EC form. Hence, both execution events and specification terms are converted to the EC format, that are directly comparable.

- **Properties of Interest:** Both functional and non-functional properties are monitored during system execution. The default WS-Agreement document contains terms for quality of published services and resources availability (non-functional constraints). To place the functional requirements into the WS-Agreement document, the EC-Assertion language is used. As denoted before, the verification framework is able to monitor the expected system behaviour, i.e. functional requirements.

- **Service Composition:** the execution environment is decoupled from runtime verification framework, both orchestration and choreography paradigm can be
followed in the literature. For instance, Hallé and Villemaire (2009 although concentrates on the choreographic constraints within business contracts, it is applicable to the web services composed by the orchestration.

5.1.3 Discussion

The calculus and logic based approaches exploit theoretically well-founded formal specification languages that enable stakeholders to define many temporal constraints and properties of interest. Employing non-intrusive monitors, while may slightly hinder the timeliness of the monitoring process, does not damage neither the execution nor the performance of the subject system. Moreover, applying widely distributed monitor instances may compensate dismal performance of the monitoring process (Mahbub, 2005).

5.1.4 Related work

Logic and Calculus family methods (e.g. LTL, EC) have mostly been applied in runtime verification process Halle and Villemaire (2010); Mahbub and Spanoudakis (2007); Cao et al. (2010). In SOSs, EC based approaches are initially introduced by Mahbub and Spanoudakis in their initiative papers. A primary runtime verification framework based on EC is presented in the Mahbub and Spanoudakis (2004b) and Mahbub and Spanoudakis (2004a) that introduces the theoretical foundation of the framework, and the extraction algorithm of EC-based requirements from BPEL. Farrell et al. (2004a) defines a formalised ontology using EC, to capture aspects of SLAs regarding utility computing (i.e. computer power, storage, network bandwidth, etc). Later on Mahbub K. (2005) an initial implementation and evaluation of the framework is experimented on a Car Rental System (CRS) case study. The case study is inherently described in Mahbub and Spanoudakis (2004a) and reused further to the research studies. Spanoudakis and Mahbub (2006) is a seminal edition of the core research work we have just discussed above. The new formal specification language, EC-Assertion, is originally presented at the last paper Mahbub and Spanoudakis (2007). In Halle and Villemaire (2010) a detailed tutorial on a demo application is represented. It illustrates the runtime verification of the properties formalised by LTL-FO+ Halle et al. (2009), which is an extended form of LTL for web services.

5.2 Workflow monitoring

Many business processes and composite services are represented by workflow applications, that form rule-based management software (McGovern et al., 2006). Runtime verification of workflow applications due to dynamically ever changing business requirements and objectives is obviously crucial. Therefore, workflow monitoring is an interesting research topic among studies which we review in this section. The core research work in this category is Dynamic Monitoring (Dynamo) (Baresi and Guinea, 2005a) in which the verification of workflows at the execution time is implemented. Dynamo verifies the evolving client-side state of the business process by monitoring services interactions.
5.2.1 Verification framework

Verification Framework in Dynamo has two principal parts: BPEL and Monitoring Manager. BPEL is responsible for cultivating the relationship between the business process execution flow and the monitoring manager, by instrumenting the BPEL document. The central component for monitoring process is the monitoring manager (Figure 5) comprising of several constituents:

Figure 5 Monitoring Manager (MM) architecture

1. **Rules Manager**: that provides necessary interface and actions for the monitor manager interactions with external components.

2. **Configuration Manager**: all queried assertions are sent to the monitor manager before business process execution. These assertions and other additional information for the verification process are properly retained in the configuration manager component. This will indeed reduce the required data for the monitoring process at the runtime, and therefore, the monitor reaction time would drastically decrease.

3. **External Analysers Manager**: the capability of using various external data analysers, which receive the monitoring data and check them against desired properties and constraints, is a distinctive feature of Dynamo verification framework. Then the external analysers manager has to format the monitoring data and the queried assertions in an acceptable structure to the external analysers.

4. **Invoker**: it is devised for invoking all requisite external components required for monitoring process, like external data collectors, data analysers, etc.

The monitor manager is actually a simple proxy server that collects the required data for the verification process, and sends it in acceptable formats to the external components. In other words, the verification process is outsourced to the external monitors that are provided by external published services. This dynamically reconfigurable framework has also enabled the data acquisition process to be driven by external web services. Monitor manager can check pre/post-conditions in the business processes by injecting appropriate monitors service invocation before/after the subject services request that create the business process.
5.2.2 Characteristics
Dynamo is a highly flexible verification framework, and many of its characteristics are potentially relevant to the activities that are performed by external services and tools; however, thanks to Baresi and Guinea (2005a), an experimental evaluation is carried out by a set of external services and tools which help examine most of the characteristics of Dynamo.

- **Monitors:** As discussed in Section 5.2.1, monitors are implemented by external web services that are coordinated by the monitoring manager. Therefore, monitors are non-intrusive; although, instrumenting the BPEL document in order to inject the monitor services invocation before/after the subject services request, manipulates the business process execution. This causes a modest degree of intrusiveness in the execution governance of the system.

  The level of distribution of monitor instances is contingent on the external monitor services deployment. There might be several monitor services, each capable of checking a specific type of constraints, so-called *External Analysers* (Figure 5). Moreover, an individual monitor service may be a compositional service per se, forming multi-fold monitor distributed deployment.

- **Monitoring Process:** The monitoring process is entirely dependent upon the external services that implement the monitors. Hence, the timeliness of the monitoring process performed by external analysers and monitors, might be whether on-line or off-line. Nevertheless, existing a separate component, i.e. the configuration management (Figure 5), that obtains the required data for monitoring only at the runtime, implies on-line monitoring mode.

  The level of abstraction of the monitoring process in Dynamo is led by the formal specification language (next characteristic), i.e. WSCoL, which allows a wide range of constraint description: from the business level to the lowest level of abstraction. For instance, in Ghezzi and Guinea (2007), the monitoring abstraction level of the case study is set at the process-level, while in Baresi and Guinea (2005a) there is a functional requirements that is verified by confirming the syntactical correctness of messages (message-level – Section 4.2.2). Again these constraints are mainly discharged by monitoring the activities which build the interactions with external services. In fact, Dynamo implicitly relies on the internal business logic and the external services.

- **Formal Specification Language:** Web Service Constraints Language (WSCoL) Baresi and Guinea (2005b); Baresi et al. (2006) inspired by light version of Java Modelling Language (JML) Leavens et al. (2006), is the constraint specification language in Dynamo. WSCoL is a standard assertion language that not only provides conventional means for accessing local variables within the process, but also supports specific features to retrieve accessed external data by collaborating with external services. The latter would help the user to define more widely constraints through both internal and external variables, and consequently, increase the expressiveness of WSCoL. Furthermore, WSCoL assertions can be fluently formatted into various representations that are acceptable to many data analysers. For instance, Baresi et al. (2006) fits <wscol></wscol> tags into a WS-Policy file format to comply WSCoL with WS-Policy framework. Later on Baresi et al. (2007a) a temporal extension of WSCoL, called *Timed WSCoL*, is proposed to place timing
constraints with common temporal operators like Becomes, Until, or Between. Obviously, the expressive power of Timed WSCoL is more than the original WSCoL.

The similar study to Dynamo in this category Lazovik et al. (2004), introduces XML Service Assertion Language (XSAL) to formally specify the desired constraints and assertions using BNF notation and propositions. Barbon et al. (2006) presents a new formal specification language, so-called Run-Time Monitoring specification Language (RTML) to specify Boolean, statistic, and time-related properties. In addition to the high level of expressiveness, there is an algorithm to automatically generate monitors from the XSAL that in turn derived from user provided XML Service Request Language, XSRL Aiello et al. (2002), which dislike Dynamo provides high degree of automation.

- **Development Language:** The main development language for Dynamo (and many similar studies in the category), is the de facto standard, BPEL. However, the original BPEL document is instrumented by monitoring services invocations to drive the monitoring process. The locations where the BPEL document has to be instrumented, are expressed via XPath expressions (Clark and DeRose, 1999) that is followed by a keyword indicating that whether the constraint is a pre- or post-condition (a newer version of XPath is also available in Berglund et al. (2003)).

- **Monitoring Data:** In the offered verification framework, acquiring data for the monitoring purposes is solely devoted to the data collector services within external data analysers. In fact, the key feature of Dynamo is distinguishing the data acquisition process from the data analysis process. As denoted before, the data can be also captured simply from external published services. For instance, in the given example in Baresi and Guinea (2005a), there is a defined post-condition which requires an image meta-data that is provided by an operation of an external service – so-called MapWS.

- **Realisation Mechanism:** Monitor external services are responsible for the realisation mechanism to perform a meaningful comparison between monitoring data and WSWoL pre- and post-conditions. Baresi et al. (2004) suggests translating the WSWoL constraints into the CLiX rules which are then passed together with instrumented BPEL to the XlinkIt (Nentwich et al., 2002) verification engine (see Nentwich, 2004) or http://www.clixxml.org/ for more details). XlinkIt produces another XML document based on the found violations when the instrumented BPEL is executing.

- **Properties of Interest:** WSWoL specification language allows to specify only functional requirements by using pre- and post-conditions on the business process execution, even though it is feasible to slightly modify Dynamo so that QoS properties are verifiable. For example, in Baresi et al. (2006) some security properties (e.g. using 3DES algorithm for message encryption) are contained in the WSWoL document which has to be discharged by the system. However, we believe that verifying QoS properties such as assuring that received messages are not eavesdropped or altered by unauthorised people, or the response message is received in a reasonable round-trip time, requires developing well-founded formal specification language, exclusively for QoS purposes.

- **Service Composition:** All services in Dynamo are orchestrated by the monitor manager to invoke both published and monitoring services. However, other similar
work in this category (Lazovik et al., 2004), checks the choreography assertions in the composition of the web services in certain business settings. Also in Dingwall-Smith and Finkelstein (2007), the choreographed service composition is checked against policy constraints which are based on Web Services Choreography Language (WS-CDL – see http://www.w3.org/TR/ws-cdl-10/) and XlinkIt.

5.2.3 Discussion

In comparison to other categories, Dynamo presents dynamically reconfigurable runtime verification framework. In particular, Dynamo proposes a monitoring framework in which all verification, data collection and analysis processes are delegated to the external services (external here is from monitoring operation perspective, from the business process point of view, these services can be intelligibly whether internal or external). This dynamic architecture however, suffers from the overwhelming number of service invocations, and consequently, the huge mass of messages exchange.

Due to non-intrusive monitor services that Dynamo applies, the timeliness of diagnosing deviations slopes down to the off-line mode. This would even worsen if the number of required pre- and post-conditions to be checked exceeds the time budget, especially in safety-critical and real-time SOSs. Besides, in the ultra-reliable SOSs that needs checking enormous number of properties and constraints, the monitor manager might become excessively overloaded, and therefore, unavailable. This problem can be resolved by assigning dedicated redundant resources to the monitor manager (e.g. widely distributing the monitor manager across the SOS) but reduces the flexibility of the SOS, as well.

Another deficiency in the current edition of Dynamo, as indicated by Ghezzi and Guinea (2007), is the potential performance bottleneck that is caused by inefficient realisation mechanism (CLiX rules). Hence, WSCoL analyser should be re-implemented by Java to alleviate the performance burden of transformation WSCoL expressions into the execution behaviour.

Although incompetence of WSCoL at defining timing constraints is overcome by introducing Timed WSCoL, specifying non-functional properties and QoS constraints in SOS, still remains uncovered by Dynamo.

5.2.4 Related work

The Dynamo basic ideas appear in Baresi et al. (2004) where the desired properties and constraints are annotated in the source of BPEL programs, and with appropriate exception handler procedure for every constraint violation. In this study, in addition to monitoring functional properties, timeout error (a non-functional concern), and runtime errors (external services failures) are also verified.

Two monitor implementation plans are also proposed by using C# and XlinkIt, respectively. In the C# implementation, exploiting the ability of .NET framework in compiling C# intermediate code at the runtime, monitors are implemented by calling create function to insert the monitor assembly code into the main code while the application is executing.

In the XlinkIt implementation, constraints are described using CLiX rule language, that in turn are validated and verified by a rule processing engine so-called XlinkIt. The latter implementation approach is further developed in the further studies (Baresi and Guinea, 2005b).
Dynamo is officially introduced in Baresi and Guinea (2005a) without presenting the formal specification language, i.e. WSCoL. WSCoL is proposed in Baresi and Guinea (2005b), and extended to be placed in a WS-Policy document (Baresi et al., 2006). Aspect-oriented prototype development of Dynamo is depicted in Baresi et al. (2007c) (also see Section 5.4). Baresi et al. (2007b) enhances business processes with self-healing capability using Dynamo (see also Section 6). Similarly, Repp et al. (2008) proposes both runtime verification and reconfiguration of service-based workflows, which is basically identical to the Dynamo, i.e. workflow execution instrumentation and separate external services for monitoring purposes.

Salayandía and Gates (2007) illustrate a road map towards workflow system management that is driven by runtime verification framework to provide more reliable business processes.

5.3 State-based conformity assessment

In the state-based conformity assessment, both the desired properties and constraints, and the execution behaviour commonly represented by service interactions, are transformed into an intermediate notion, i.e. state-based models like automata, for facilitating direct comparison. In other words, behavioural specification is represented by (or converted to) a state-based rendition, on one side, and on the other side the execution behaviour are also represented by (or converted to) an identical state-based rendition. The monitors then can simply check the execution model to demonstrate conformity with the specification model, because they are both expressed in the same language model (e.g. automata). The core studies in the category are two research work: Simmonds et al. (2009) and Li et al. (2006).

5.3.1 Verification framework

The verification framework of the research category comprises of three main components (Figure 6).

**Figure 6**  The verification framework of Simmonds et al. (2009)’s work
1. **Property Manager**: which is responsible for receiving the desired properties and constraints couched in the user-defined language, and converts them to the intermediate language model. In fact, the property manager is the front-end component of the verification framework in which the user interface and all other user interaction tools are provided.

2. **Message Manager**: since in many studies in the category the execution behaviour is represented by observable service interactions, there should be an integral component in the framework to collect exchanged messages between services. The message manager obtains relevant messages for the verification, and sends them to the central component of verification framework, i.e. monitoring manager.

3. **Monitoring Manager**: the central part of the verification framework in the category is the monitoring manager which has to create and manage monitors, and nurture them with relevant events according to the monitoring properties. The conformity procedure is conducted by both monitor instances and the monitor manager, and the potential dissociated deviations are reported to the user via the property manager component.

Simmonds et al. (2009) exploits another component so-called *message handler* which acts as an interface between BPEL execution engine and the verification framework. Li et al. (2006) also uses an auxiliary component in the level of constraints named *Constraint Level Validation Manager* (CLVM), that is created by another component called *Service Level Validation Manager* (SLVM), directly analogous to monitors and the monitoring manager, respectively.

### 5.3.2 Characteristics

There are two core research studies in the category, and there are differences between some of their characteristics; however, there still exists significant similarities. We strive to present both similarities and dissimilarities in a comparative manner.

- **Monitors**: Using a message manager in the verification framework (Figure 6) shows that non-intrusive monitors are employed. In particular, the constructed monitors observe generated events by the system execution engine, and thus, there is no direct access to BPEL source document.

  Assigning an individual monitor instance to every service interaction constraint, requires multiple instantiations of monitors. Nevertheless, there are not any obvious clues indicating the level of monitors instances distribution, neither are there any implicit assumptions about centralised deployment. As a result, the monitor instances can be deployed in whether a distributed or centralised manner; though, a distributed deployment seems more reliable.

- **Monitoring Process**: All relevant messages are automatically forwarded to the corresponding monitor instances by the monitoring manager. The monitoring process is conducted through conformity procedure where execution *trace automata* are checked against the *desired automata* which are extracted from formal system specification models. Incompliant states/transitions simply represent deviation from system requirements. If every encountered states/transitions in the trace automata are constantly compared to the desired automata, then the conformity procedure tends
towards on-line monitoring mode. Note that the trace automata are implicitly constructed during execution, while the desired automata are usually generated before system execution. The timeliness of the monitor manager also affects the point of monitoring mode in the off-line/on-line spectrum. Since the messages for the verification purposes are commonly immediately sent by the message manager, the monitoring mode is often on-line.

In addition to monitoring the lowest level of abstraction, i.e. individual SOAP messages, the monitoring process might be also driven to higher levels of abstraction namely the message-level. The reason is that the interactions between services lead to monitor not only the message contents (low-level) but also the sequence of messages (message-level). Generally speaking, the abstraction level of the state-based models determines the abstraction level of the monitoring process.

- **Formal Specification Language**: Studies in the category use various formal specification languages; although, all of them are transformable to state-based modelling languages. Li et al. (2006) builds an ontology for Interaction Property Patterns (IPP) (Li et al., 2005) on the Specification Pattern System (SPS) (Dwyer et al., 1999) as an add-on to the Ontology Web Language for Services (OWL-S – see http://www.ai.sri.com/daml/services/owl-s for more information). Desired properties and constraints in IPP are formally defined in Finite State Automata (FSA) (the intermediate language for conformity procedure). The runtime behaviour of the system then advances the FSA according to the intercepted messages (Li et al., 2005).

On the other hand Simmonds et al. (2009) presents more expressive formal specification language to specify web service conversations that are selected from prevalent UML notions collection, i.e. Sequence Diagram (SD). Since SD is a semi-formal modelling language, it should be formalised by being converted to Non-deterministic Finite Automata (NFA), and also applying proper SPS operators. Leveraging industrially widespread specification models in UML like SD, provides high degree of expressiveness in specifying properties and constraints. Moreover, there are many available tools expediting monitor construction (Gan et al., 2007), and also extraction of properties and constraints from specification models.

In Dranidis et al. (2009) web services constraints are directly specified by a state-based language called Stream X-Machine (SXM) language which uses JSXM (Dranidis, 2009) XML-based syntax, and thus, appropriate to SOS applications. Aalst et al. (2008) exploits another type of state-based modelling language so-called Petri-Nets, that is represented by Petri Net Markup Language (PNML). In fact, BPEL documents are automatically translated into PNML, and further converted to the Workflow Nets (WF-Nets) which are suitable for the conformity procedure at the execution time.

- **Development Language**: Non-invasiveness nature of state-based monitoring allows developers to choose an arbitrary development language. In other words, the verification framework interacts with the monitored system via only events that are represented by the exchanged messages. Hence, there must be only a full understanding of service communication protocols and the message contents format (e.g. SOAP). Moreover, the development language should be devised so that the
execution trace can be quickly transformed into intermediate state-based models. For instance, Raimondi et al. (2008a) employs timed automata (TA) as a simple way to model the execution trace at the runtime, and checks them against TA-specified non-functional constraints. However, that does not make sense to construct monitors disregard of the overlaid SOS development languages like BPEL (Simmonds et al., 2009).

- **Monitoring Data**: Most of studies in the category monitor the interactions between services, and thus, acquire only relevant data to the service interactions. This means that specific components (e.g. message manager) collect those messages that reflect only the service interactions, and then send them to the corresponding monitor instances via monitoring manager.

  In Li et al. (2006) data are acquired from both low-level sources (message contents) and middle-level sources (message sequences) indicating communications between multiple web services. Conversely, Simmonds et al. (2009) has rather high-level sources for acquiring data. Only events are captured that are relevant to the services invocation within the execution of a single business process. Aalst et al. (2008) identifies higher-level data sources where the messages are grouped into log traces that represent business processes execution. Log traces are abstracted to form labels that are further used by a conformity procedure (labels create the transitions in the state-based model, indeed).

- **Realisation Mechanism**: The conformity procedure, which is created for matching the specification state-based model with the runtime generated state-based model, form the realisation mechanism. There is an individual monitor instance for every specified property which is automatically handled by the monitoring manager. In fact, each monitor instance corresponds to a specification state-based model indicating the desired system traits. Monitoring manager is responsible for creating monitor instances by extracting state-based models from user-defined constraints. Therefore, a small part of the realisation operation that links formal specification models to the execution-generated models is indeed regulated by the monitoring manager.

- **Properties of Interest**: Applied specification languages in most of studies in the category, restrict the verification process to monitoring only functional requirements. However, Simmonds et al. (2009) enriches the proposed specification language (i.e. SD) by using specific operators to model existence and non-existence scenarios for liveness and safety properties, respectively. On the other hand, whereas Li et al. (2006) mainly introduces the verification framework for monitoring functional requirements, it is plausible to implicitly specify non-functional concerns by discharging placed constraints upon functional aspects of the system, e.g. defining a boundary on the number of service invocations (functional aspects) to achieve the defined performance parameter (non-functional concerns).

Raimondi et al. (2008a) presents a methodology to specify non-functional concerns for web services through TA, and verify them during execution time by checking the acceptance of a word, which represents service execution trace, within the TA model.
Service Composition: Like many other non-intrusive approaches, state-based conformity verification framework is separated from business process execution engine, and thus, is adjustable to embracing both orchestrated and choreographed paradigm. For instance, Aalst et al. (2008) introduces the runtime verification framework within a choreographic service composition paradigm.

5.3.3 Discussion

To some extent state-based conformity assessment is similar to logic and calculus based approaches reviewed in Section 5.1, as they both propose non-intrusive monitors which scan events generated by the execution engine. A non-intrusive approach although benefits less performance effects on the main process execution, becomes entirely futile in real-time systems due to significant delay in diagnosing deviations.

Another disadvantage of the category is that the desired properties and constraints still have to be explicitly specified by the user, and thus, the potential inconsistencies in the user-defined specifications are underestimated. Therefore, there have to be clever tools or gadgets to automatically extract consistent constraints from the system specification. In Aalst et al. (2008) for example, two tools are introduced to convert abstract BPEL documents to formal Petri Net models.

Simmonds et al. (2009) concludes that current used state-based models are not applicable to monitoring data, and hence, other formal specification models are required, e.g. Parameterised NFA (PNFA) Baier and Mcilraith (2006) or Colored Petri Net (CPN) models Jensen (1991) that support data representation.

Most of studies in the category mostly concentrate on the relatively low-level abstraction, and more high-level verification (e.g. business processes monitoring) still remains open to future studies. A premature research report in Bratanis et al. (2010b) suggests Business Level Agreement (BLA – see Section 3.4) to support high-level business contracts by exploiting SXM formalism. The proposed approach is similar to the core research work discussed in Section 5.2 in deploying monitor instances as individual web services. Aalst et al. (2008)’s approach has also supported monitoring high-level business requirements, but in all of them verifying non-functional properties are not specifically addressed.

5.3.4 Related work

Broad research agenda for the core research work Simmonds et al. (2009) is established in Gan et al. (2007). Later on in Simmonds et al. (2008) SPS method is applied to model temporal logic property patterns through SD templates. Self-healing extension of the core research work is elucidated in Simmonds et al. (2010a) (see also Section 6.1). In Simmonds (2011) SD modelling language is extended to a fairly independent and expressive language called Web Sequence Diagram (W-SD) with various developed recovery strategies. More detailed description of Li et al. (2006) has been already presented in Li et al. (2005). A novel tool for web services compositions monitoring based on the sequence charts, so-called WS-PSC, is described in Zhang et al. (2010).

Dranidis et al. (2009) exhausts a special computational model capable of modelling both the data and the control of a system called SXM. The proposed approach is identical to the state-based conformity assessment explained in this section, unless it is competent to be used in consumer, provider or even broker SOS roles (see Section 3.1). The last
work in the category is Bratanis et al. (2010a) in which it employs SXM to model and verify the conversational web services. Pursuing this line of research, Bratanis et al. (2010b) introduces business-centred monitoring framework whereas Aalst et al. (2008)’s approach relies on earlier studies in the process mining van der Aalst and Weijters (2004).

5.4 Aspect-oriented verification

Aspect-Oriented Programming (AOP) finds crosscutting concern points within the code that should be monitored during program execution, and mechanises weaving advice code which must be executed instead of the vulnerable code. In this category, the studies are explained that employ AOP in the runtime verification of SOSs Seyster et al. (2010); Avgustinov et al. (2007); Shin et al. (2007); Avgustinov et al. (2006); Narendra et al. (2007); Huang et al. (2009); Kallel et al. (2009); Courbis and Finkelstein (2004). Nonetheless, the core research work in the category is Huang et al. (2009) which presents a generic framework that is applicable to most of AOP approaches.

5.4.1 Verification framework

Aspect-oriented runtime verification framework firstly requires a development environment for expressing crosscutting points and advices. Since AOP also manipulates the way of system execution, there must be an AOP-customised execution engine to find the crosscutting points and weave advices into the code, if necessary. Therefore, there often exist a runtime aspect extension which enables these activities during system execution (Huang et al., 2009).

Figure 7  Aspect-oriented verification framework

In the following, the principal components of the aspect runtime extension that act as the runtime verification framework for AOP systems, are listed (Figure 7):
1 **Aspect Registration:** which provides a dynamic repository for all defined aspects. In fact, all information on the matching point-cut expressions and the associated events are stored in the entries of aspect registry.

2 **Monitor Manager:** that is responsible for handling generated instances of monitors and the relevant events. It might have to maintain an events tree to properly route events to the correspondent monitor instances.

3 **Aspect Manager:** which is the central manager part in the AOP verification framework. The monitor manager is invoked by the aspect manager. It is also responsible for weaving advices code into the process execution code, if applicable.

Just like other aforementioned categories, AOP runtime verification framework demands AOP-customised execution engine to provide necessary events for the verification framework.

5.4.2 **Characteristics**

There are several work in the category, but we strive to concentrate on the core research work in Huang et al. (2009), meanwhile relative comparisons with other studies are drawn.

- **Monitors:** Depending on the specified properties and constraints, certain program locations are identified—so-called join points—which demonstrate the places where an aspect becomes enabled and the corresponding monitor instance has to be invoked. For example, Huang et al. (2009) employs an individual monitor instance for each specified aspect so that if an event associated to the defined aspect is occurred, the respective monitor will be invoked. Since aspects are usually specified within the subject code, the level of monitor intrusiveness is relatively high Narendra et al. (2007); Huang et al. (2009); Kallel et al. (2009); Courbis and Finkelstein (2005).

  The number of monitor instances and the degree of distribution varies depending on the number of aspects, and the number of monitoring aspects in every instances. Note that, respecting to the verification framework (Figure 7), making and removing monitor instances are handled by the monitor manager (Huang et al., 2009).

- **Monitoring process:** Many studies in the category propose using AOP to usually check timing constraints (non-functional concerns) imposed upon the system (Kallel et al., 2009; Narendra et al., 2007). This often leads to an on-line monitoring process with a reasonable diagnosis time. In fact, aspect-oriented verification benefits from on-line monitoring, and consequently real-time diagnosis at the cost of increasing the degree of intrusiveness, and hindering the subject program performance.

Due to the existence of an aspect-oriented implementation of BPEL, so-called **AO4BPEL** Charfi and Mezini (2007) that is used in many studies in the category Huang et al. (2009); Courbis and Finkelstein (2004, 2005), the monitoring process is driven in the process-level of abstraction. For instance, Huang et al. (2009) mainly focuses on the level of WS-BPEL abstract, i.e. the process-level, even though various instances of the process is monitored. However, if monitoring various instances of business processes – that satisfy a business objective when integrated–
are handled systematically, the monitoring process will be driven at the highest level, i.e. the business-level (Section 4.2.2). Courbis and Finkelstein (2004)’s approach provides three different levels of abstraction for the monitoring process: (1) the semantic analysers which is identical to the business-level, (2) the BPEL engine that is devoted to monitor the execution engine per se, (3) the business processes executed by the engine, i.e. process-level.

- **Formal Specification Language (AOP):** is actually a programming paradigm that is closer to the implementation of the system, and thus, many studies employ aspect-oriented development language to specify the desired properties and constraints (e.g. Huang et al., 2009). However, many other studies introduce novel formal specification languages that is modified for aspect-oriented development. For instance, Kallel et al. (2009) introduces Extended Time Unit System (XTUS) combined with TA to expressively specify temporal constraints. In particular, XTUS is proposed to remedy the deficiency of using absolute time in specifying temporal properties, which is not supported by TA. Besides, XTUS is straightforwardly translatable to aspect templates. The syntax and semantics of XTUS is inspired by Z notations (Potter et al., 1996).

Narendra et al. (2007) employs a SOS-based specification language combining WS-Policy and Web Service Level Agreement (WSLA) documents (Keller and Ludwig, 2003) which is dedicated to specify only non-functional concerns. WS-Policy is the main framework for specifying the requirements of the web services, while WSLA is only a standard schema to define the expected behaviour of the web services. An XML file is also used to represent aspects and their interactions within WS-Policy and WSLA.

Huang et al. (2009) uses XPath and necessary operators to specify historical point cuts and combine them, respectively. The hallmark of Huang et al. (2009) stems from stateful aspects where user can define the desired properties and constraints using historical events occurred during system execution. Moreover enabling describing pattern of interests (instead of a single property or constraint) reduces the number of monitoring events at the runtime.

- **Development Language:** Since aspects are often expressed by the locations within the executable codes, aspect-oriented verification is tightly coupled with the development language. A lot of studies in this category exploit AO4BPEL (Charfi and Mezini, 2007), which is one of the most prominent aspect-oriented development languages for the web service compositions (Huang et al., 2009; Kallel et al., 2009). However, AO4BPEL restricts the verification process to the BPEL documents and process-level monitoring. Lower levels of monitoring still requires more investigation into other low SOS-based development languages (e.g. SOAP).

- **Monitoring Data:** As denoted before, the majority of studies in the category contemplate process-level monitoring, and accordingly the monitoring data are at the process-level. Irrespective to the type of the desired properties and constraints (e.g. non-functional properties Narendra et al., 2007 or general properties Huang et al., 2009), monitoring data usually represent consequences/timeliness of services invocations. For instance, Huang et al. (2009) illustrates an on-line travel arrangement process motivating example in which an aspect-oriented verification
process is constructed to check that before fulfilling a user’s trip cancel request, both hotel and flight reservations must be prosperously cancelled. Again acquiring lower levels (e.g. service-level, message-level or SOAP-level) of data for monitoring is not specifically addressed by the studies in the category.

- **Realisation Mechanism**: As the aspect-oriented verification is performed in the system implementation level, the realisation mechanism constitutes multi-layer procedures to bring formal requirements specification into the crosscut expressions. XML-based intermediate languages usually form middle layers that facilitate transforming specifications into the aspects. Kallel et al. (2009) uses XTUS and TA as intermediate languages to specify desired properties and constraints by inserting them into the BPEL document using XPath expressions. Huang et al. (2009) follows the same approach by just using XPath operators without employing intermediate languages. Respecting applying intermediate languages, the work within the category is roughly similar to the state-based conformity assessment category discussed in Section 5.3. In both categories, using higher level specification languages, and accordingly, shifting the realisation mechanism towards abstract imperative languages remains unresolved.

- **Properties of Interest**: Generally speaking, AOP is a paradigm to localise functional requirements that cut across multiple software modules. As a result, many proposed approaches monitor and verify the functional properties ranging from temporal constraints placed upon the functional behaviour of the system (Kallel et al., 2009), to the normal features that the behaviour of the system has to retain (Huang et al., 2009).

Nevertheless, Narendra et al. (2007) presents a runtime verification of non-functional properties (e.g. response time) by properly equipping WS-Policy documents grammar with NonFunctional-Property tag, and enabling users to specify the desired non-functional concerns.

- **Service Composition**: To the best of our knowledge, current research work in the category focuses on aspect-enabled business process execution to verify it, and also provides alternative execution sequence if applicable. Therefore, the verification framework usually uses an orchestrator as the execution engine, to find crosscutting points as well as to weave the advices if necessary. For instance, Courbis and Finkelstein (2004) employ an aspect-enabled BPEL execution engine to implement aspect-oriented verification process and healing ritual for the centralised web services composition.

### 5.4.3 Discussion

AOP, per se is a predominant paradigm in runtime verification process regulation. From SOS perspective however, due to high dependency to the development language of the subject system, it violates the platform-agnostic principle of SOS. In other words, the desired properties and constraints that have to be checked using AOP, must be expressed by the development language. To overcome this drawback, many studies only allow developers to define aspects within the BPEL documents. Consequently the executor should be modified so that it can find crosscutting points and weave the advices. However, as discussed in Section 5.4.2, this solution enforces the developers to define
only high-level (often process-level) desired properties and constraints. In some domains, more refined monitoring level is required and so investigating a reasonable and pragmatic aspect-oriented verification approach remains as a potential line of research.

Noteworthily, AOP retains a salient feature enabling the verification framework to contain recovery statements (see Figure 1) by weaving advices into the vulnerable code. The woven code is executed whenever a violation is diagnosed in the join points. Hence, aspect-oriented verification can be further extended to a dominant approach in constructing self-healing systems (see Section 6.1).

5.4.4 Related work

AOP is an increasingly popular paradigm among runtime verification community Sokolsky (2007), but from SOS point of view, it is fairly a novel technique. One of the earliest studies regarding aspect-oriented paradigm in SOS is appeared in Cibrán and Verheeecke (2003) which applies AOP to modularly manage services issues by introducing Web Service Management Layer (WSML). Verheeecke and Cibrán (2004) presents a similar approach implemented in JAsCo (Suvee et al., 2003), a dynamic AOP language using Java Beans component model. Stateful aspect-oriented programming is originally presented in Braem and Gheysels (2007) using Padus, an AOP language for WS-BPEL, which is later followed by Huang et al. (2009) to introduce AO4BPEL. Mostéfaoui et al. (2006) employs AOP as a flexible framework that is adaptable to changing requirements for security purposes in web services. This work is indeed the seminal edition of Narendra et al. (2007) which is discussed in the category.

An aspect-oriented prototype of Dynamo (Section 5.2) is proposed in Baresi et al. (2007c). The work is then extended by Baresi et al. (2007a) using Timed WSCoL so that both functional and non-functional properties and constraints can be verified. A survey of aspect-oriented solutions into the problem of non-functional properties formal description for web services using Web Service Modelling Language (WSML – see http://www.wsmo.org/wsml/wsml-syntax), is presented in Ortiz et al. (2004).

5.5 SLA-driven compliance

One of the main purposes of SOS is to create an open, integrated, inter-operable infrastructure for composing services from various external providers, and to build business processes (see Section 3.3). Third-party service providers should fulfil contracts with the consumers to satisfy formally defined soft goals and avoid financial penalties. Service-Level Agreements (SLA) is the formal definition of contracts between external providers and the consumers, which should be complied at the runtime (see also Section 3.4). In this category, we introduce the research work which performs runtime verification process to check runtime system behaviour against SLAs. The core study is Raimondi et al. (2008b), and other similar studies are only explained when dissimilarities are notorious.

5.5.1 Verification framework

The verification framework in the category is almost identical to the illustrated generic runtime model (Figure 1) except that the desired properties and constraints are expressed in the form of SLA documents (Figure 8).
Nonetheless, in order to preserve paper monotony, we briefly describe every component of the framework as follows:

1. **SLA Document**: represents the formal standard contract between third-party service providers and the consumers which describes usually the quality of provided services (or QoS- Section 3.4). The format of an SLA ranges from informal contracts to predefined templates or XML-based documents like Web Service Level Agreements (WSLA) (Keller and Ludwig, 2003).

2. **Formal Specification**: represents formalised format or an intermediate formal model (regularly state-based models) of an SLA document. Formalisation is essential for a trustworthy runtime verification process (see Section 2.3).

3. **Execution Trace**: represents all mandatory data acquired for verification process at the execution time which illustrates the execution sequence of business processes. Respecting the intermediate formal model, monitoring data are usually represented in an appropriate format.

4. **Runtime Checker**: or monitor that is responsible for checking runtime service behaviour against formal specification models.

**Figure 8** Runtime verification framework using SLA

Most of the current work transforms SLA documents into formal state-based models which can be simply checked against runtime-generated counterpart models. Hence, to some extent many studies in the category are similar to state-based conformity assessment category, discussed in Section 5.3.

5.5.2 **Characteristics**

However, the characteristics of the current studies in the category have resemblance to studies within other categories; we examine them according to the SLA documents.

- **Monitors**: As explained in Section 5.5.1, monitor is a runtime checker which is responsible for comparing the desired properties and constraints with service runtime behaviour. Analogous to a generic monitor, SLA-driven monitor is usually a stand-alone (non-intrusive) program which receives events as the system execution proceeds, and performs an state-based comparison (Raimondi et al., 2008b). However, in Zulkernine et al. (2008) the level of non-intrusiveness is expanded to several specific levels: the internal-level where monitors are deployed as internal
Runtime verification of service-oriented systems

agents within the messaging framework, and the external-level where monitors are deployed as external agents intervening between third-party published services and the consumers (both levels are still independent of the subject service with different levels of intrusiveness). The former provides more flexibility and less messaging overhead, while the latter allows easy maintenance at the cost of dismal system performance.

The distribution degree in which the monitors are deployed, is fully adjustable to the level of monitors intrusiveness, and also the implementation of monitor instances.

For instance, Zulkernine et al. (2008) employs a centralised monitor –so-called Performance Monitor (or PM)– to acquire required data from SOAP messages for SLA monitoring. Once web services start executing, upon receipt of SOAP messages, they will send the relevant necessary data to PM; PM then validates the compliance of received messages to SLA. Note that the proposed approach is actually a middleware for verifying web services compositions that centrally outsources the monitoring process to the distributed monitor instances by PM.

• Monitoring Process: The monitoring process is usually closer to the on-line end of timeliness spectrum (Figure 3): on-time generated events are passed to the monitor instances in a timely manner (Raimondi et al., 2008b). Nevertheless, due to the disproportionate amount of generated events and inefficient monitor instances implementation (to achieve flexible verification framework), monitoring timeliness may be degraded towards off-line monitoring mode, like Zulkernine et al. (2008).

The abstract level of monitoring process is in lower levels, i.e. it is based on the captured low events (e.g. SOAP messages Raimondi et al., 2008b). Whilst Zulkernine et al. (2008) proposes a process-level monitoring framework, the core of monitoring process is primarily driven by SOAP messages. This enables twofold abstraction level, which provides sufficiently flexible verification framework.

• Formal Specification Language: There is a wide range of formal intermediate models, in order to bring formality to the SLA contracts. Web Service Level Agreements (WSLA) (Keller and Ludwig, 2003) is an XML-based specification framework which provides definition and monitoring of SLA for web services; although, it can be extended to other similar concepts like business processes. Three-section structure of WSLA, i.e. parties, service description, and obligations, while allows users to flexibly specify the desired SLA with various metrics, resolves inherent ambiguity in informal conventional SLA expressions.

Raimondi et al. (2008b) suggests a multi-level specification language. Simple SLA contracts are initially expressed by SLAng, a language for defining service level agreements Lamanna et al. (2003). SLAng specifications are then reduced to the Timed Computation Tree Logic (TCTL) formula, a temporal logic notion which permits time intervals Alur et al. (1993). Finally, TCTL formulas are translated to TA Alur and Dill (1994) which can be accordingly checked against execution traces that are represented by similar state-based models.

Farrell et al. (2004b) presents a formalisation for the XML-based contract language so-called Contract Tracking XML (CTXML), in which contracts consist of one or more contract norms, zero or more parameters and variables. This work offers an
expressive specification language for SLA in terms of contract events, states, obligations and privileges. The semantics of CTXML is inspired by Event Calculus (EC) which is nested into the XML schema, so-called ecXML. In particular, the proposed specification language is twofold: in the high level specification, SLA contracts are expressed by CTXML, and in the low level specification, CTXML models are elaborated by ecXML. Two ways of implementation are advanced for query interpretation and formalisation of SLA: Prolog-based and Java-based implementations which both appreciate semi-automate the process of SLA interpretation and formalisation.

- **Development Language**: Similar to other categories, BPEL is the most frequently used development language in the research of this category (Raimondi et al., 2008b). This is attributable to the fact that SLA typically specifies constraints upon composite web services, expressed by BPEL. However, Zulkernine et al. (2008) employs SOAP messages as low level development language: the required data for checking SLA compliance is injected into SOAP messages by the Performance Monitor. Web services also send the requested data using SOAP messages, and in a deviation case, PM invokes alternative web services by sending proper SOAP messages. Nevertheless, specifying and executing business processes containing several composite web services, still require high level development language like BPEL.

- **Monitoring Data**: The required data for checking SLA-driven compliance is greatly dependent on the definition of SLA and the desired properties and constraints. In most studies, the essential metrics that may be binded at the runtime are specified by SLA parameters (Keller and Ludwig, 2003; Farrell et al., 2004b; Raimondi et al., 2008b) which help to identify the required data and also the abstraction level of data acquisition for the monitoring process. For example, Keller and Ludwig (2003) defines two SLA parameters for the operation getQuote of a web service: (1) AvgThroughput which determines the average throughput for an operation of the service and (2) OverUtilisation which indicates the proportion of time in which the service is over utilised based on the contract. Specific functions then have to be declared to measure these two parameters by acquiring the required data (e.g. time spent, number of transactions, etc.) from execution-generated events.

- **Realisation Mechanism**: The realisation mechanism often is a multi-level procedure that transforms SLA documents into the desired properties and constraints which are observable from execution traces. Keller and Ludwig (2003) exploits two services, measurement service and condition evaluation service, to map SLA parameters to executing entities. The measurement service is responsible for probing, intercepting, collecting and maintaining the relevant data to the SLA parameters. The condition evaluation service is given the monitoring data by measurement service, and carries out the compliance checking procedure using the thresholds defined in the SLA document. Deployment service operates at the top of measurement and condition evaluation services to decompose WSLA documents into the parts with respect to the measurement and condition evaluation services. The realisation mechanism in Raimondi et al. (2008b) is similar to the state-based conformity assessment method described in Section 5.3 where two state-based models are compared.
Properties of Interest: Unlike WSDL which describes the functionalities that a web service provides, SLA documents are used to specify non-functional properties that are agreed between external service providers and the consumer parties (see Section 3.4). This is why in some of the proposed approaches in the category (Zulkernine et al., 2008; Farrell et al., 2004b), SLA monitor is renamed performance monitor. More specifically, Farrell et al. (2004b) employs methods and techniques to specify, measure, monitor, and verify timing constraints.

Service Composition: Governing services composition is arbitrarily controlled by SLA-driven compliance verification framework, but since SLA are often used to monitor the quality of involved web services in a business process, an orchestrated composition governance is relatively more frequent (Keller and Ludwig, 2003; Raimondi et al., 2008b). Nonetheless, in some other studies (Zulkernine et al., 2008) choreographed composition governance is within the realms of possibility.

5.5.3 Discussion
Checking the SOS behaviour against SLA is directly influenced by the definition of SLA contracts, and also monitorability of the system model. An advanced technique for both modelling the system and defining SLA contracts in a three-role Application-Service Provision (ASP) scenario is presented by Skene et al. (2007).

SLA-driven compliance of web services is obviously crucial in the applications that pursue soft goals and strict performance constraints (e.g. in cloud, grid, or P2P systems). For instance in Farrell et al. (2004b), a performance monitoring framework is introduced for Utility Computing (UC) systems. Whenever a violation against SLA contracts is diagnosed, a financial penalty is often prescribed for external service providers. Hence, the verification framework may be also responsible for automatically calculating applied penalties.

Note that the verification of the system functionality can not be entirely obtained by the SLA-driven approach, and there should be similar methods to check functional behaviour correctness and consistency. In fact, SLA-driven compliance examination is usually employed as an additional approach alongside other verification techniques for checking functional requirements of SOSs.

5.5.4 Related work
An SLA-driven infrastructure for web services management and monitoring is proposed in Momm et al. (2007). It is a seminal study for the core research work Zulkernine et al. (2008) discussed in the category. Several other studies were conducted to model and monitor contracts at the runtime and the negotiation process. In Daskalopulu (2000) contracts are formalised using Petri Net models that facilitate both contract drafting and performance monitoring. Another formal modelling of contracts are presented in Marjanovic and Milosevic (2001) which includes specification, verification, modelling and scheduling of contracts, and also the required visualisation concepts for enhancing understandability during negotiation. Grosof et al. (1999) presents a representational approach consists of generalised version of Courteous Logic Programs (CLP) that handles prioritised rule conflicts in the contracts, and an XML encoding of CLP, so-called Business Rules Markup Language (BRML), in order to apply CLP in a heterogeneous environment like SOS. Cremona Ludwig et al. (2004), stands for Creation and
Monitoring of Agreements, is the matured edition of the research work Keller and Ludwig (2003), and proposes an architecture for WS-Agreement implementation previously described in the category.

5.6 Miscellaneous studies

To keep the paper genuinely inclusive, miscellaneous studies that are not discussed in either categories, are briefly introduced in this section.

A significant portion of studies in the literature is dedicated to the model-driven runtime verification Ta’id Holmes1 et al. (2010); Holmes et al. (2011); Qianxiang et al. (2007). For instance, Ta’id Holmes1 et al. (2010); Holmes et al. (2011) are two subsequent work proposing MOdel-aware Repository and Service Environment (MORSE) that provides model evolution, model-driven runtime verification, and more importantly root-cause analysis for the adaptation process for avoiding potential violations. Qianxiang et al. (2007) suggests a quality model which examines five kinds of events: response to the client, e.g. availability, efficiency, etc.; application execution, e.g. business logic, data related issues, etc.; resource state changing, e.g. CPU usage, memory usage, etc.; request message that checks the validity of requested message, or allowed number of requested messages according to the contract; and management operation which involves activities that are performed throughout the system, like reconfiguration procedure. Every event class requires specific probes mechanisms and metrics that are essential for the runtime verification of non-functional properties of SOSs.

Robinson (2003) presents a goal-oriented approach for runtime verification of web services. In KAOS approach (a requirements engineering technique, stands for Keep All Objective Satisfied), Dardenne et al. (1993) to discover the potential obstacles, monitors are automatically derived from obstacle models, and consequently monitoring web services are constructed according to the requirements models. In fact, Robinson (2003) explicitly defines the realisation mechanism by establishing direct relationship between design-time models (in KAOS) and run-time ones.

Lastly, Cao et al. (2010) although proposes a passive testing method for web services behaviour security compliance, it can be also applied to both off-line and on-line runtime verification. The employed formal specification language called Nomad is an action-based method for specifying desired security rules (e.g. permissions, obligations, etc.) within the web services. The proposed approach is then implemented as a tool so-called Runtime Verification engine for Web Services (RV4WS), that automates runtime verification of security properties.

6 Final remarks and future directions

The nature of service-oriented ecosystem is that service discovery, binding and execution are all handled at the runtime. In fact, it is impossible to determine the system behaviour prior to the execution time. Therefore, due to the intrinsic dynamism of SOS as well as the loosely-coupled architecture enabling developing heterogeneous distributed systems, conventional verification techniques, like statical approaches or testing, become grossly inadequate in service-oriented environment.
Although testing is an essential dynamic verification activity for developing many of software products; it can not fulfil the expected degree of assurance that should be provided by the software verification process, specifically in ultra-reliable domains. On the other hand, statical verification techniques like theorem proofs, model checking, etc. while providing certain degree of assurance, suffers from lack of information on unexpected situations created due to runtime behaviour, the operating environment, or the user inputs. Runtime verification is a novel dynamic verification approach in which the system behaviour is checked against formally specified constraints when the system is on-the-fly, i.e. it combines testing and formal verification.

To verify SOS, runtime verification is a comparatively trivial approach which not only monitors unexpected situations at the runtime, but also makes high level of assurance by employing formal specification languages. This approach leads to an SOS brimming with ultra fault-tolerant features.

There are often eight basic characteristics, when considering runtime verification in SOSs:

1. **Monitor** is the central element of the runtime verification framework, and may differ in level of distribution respecting to the deployment in SOS, and the level of intrusiveness regarding the subject system.

2. **Monitoring Process** can be on-line or off-line, i.e. reacts to the violations in a timely manner or not, which exhibits two ends of the wide range of monitoring timeliness. The level of monitoring abstraction can be set in four different levels which determines both procedural and data abstraction levels for the monitoring process.

3. **Formal Specification Language** is the most theoretical aspect of the runtime verification process to formally specify system behaviour properties and/or the desired constraints.

4. **Development Language** demonstrates the the other aspect of the runtime verification process, namely the implementation of the subject system.

5. **Monitoring Data** is profoundly influenced by monitoring process and the data abstraction level.

6. **Realisation Mechanism** delineates most intellectual challenge of the runtime verification process: how to check execution traces against formally specified properties and constraints.

7. **Properties of Interest** describe what types of properties, i.e. functional requirements or non-functional concerns, are checked throughout the runtime verification process.

8. Finally, **Service Composition** indicates that whether an orchestration or a choreography paradigm is embraced for governing services composition in the verification of SOS.

We also reviewed major research work in the literature that forms the following broad categories. The logic and calculus oriented approaches, workflow monitoring, state-based conformity assessment, aspect-oriented verification, and SLA-driven compliance. An overview of all research categories based on the most highlighted characteristics is shown in Figure 9. The studies generally address the realisation mechanism, which is the key challenge in the runtime verification process. Most of studies also include new or
modified definition of formal specification languages to clearly specify the desired properties and constraints. Almost half of them characterise deployed monitors, monitoring data and the service composition in SOSs. We believe that these three characteristics have to be explained in detail and further elaborated both in the current and future studies. A few of studies explicitly address the monitoring process, i.e. timeliness and also the abstraction level, which should be completely elucidated in the future work. To conclude, there is not a comprehensive yet applicable approach in the runtime verification of SOS, which devises all introduced characteristics (or probably more) to build a flexible and also dependable verification framework for many of current domains that apply SOSs.

Figure 9   Comparing salient characteristics in the research categories

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic &amp; Calculus Oriented Approaches</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workflow Monitoring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State-Based Conformity Assessment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aspect-Oriented Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLA-Driven Compliance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.1 Towards self-healing and self-adaptive systems

Considering runtime verification generic model (Figure 1), the most distinctive feature of runtime verification stems from the adaptation processes where the automated post-activities are coordinated next to the deviation detection (see also Section 2.4). In other words, the system should be enabled to automatically recover from misbehaviour and failures by conducting pathological analysis. This approach leads to the self-healing ability which provides utterly reliable systems.

Many promoting studies in the context involve addressing intractable problems in the self-healing systems. Friese et al. (2005) introduces a Robust Execution Layer (REL) for BPEL4WS execution engine that supports self-healing execution of business processes by bringing in a replacement for the defected services. To this end, the proposed middleware has to also provide an appropriate P2P service discovery procedure (Section 3.2) to enable finding alternative services that can be substituted for the defected service. Wu et al. (2009) develops a broad classification of both main faults in service-oriented ecosystem, and the correspondent recovery strategies.

A quality-driven self-healing approach for AJAX-based web applications is presented in Naccache (2007) in which the self-healing ritual is evolved out of Queuing Network (QN) models. A lightweight service management middleware, so-called Manageable and Adaptive Service Composition (MASC), that implements a set of recovery policies for reliable web service composition, is addressed at Erradi et al. (2006b). A self-healing edition of Dynamo, discussed in Section 5.2, is presented in Baresi et al. (2007b) using described concepts in both workflow monitoring and aspect-oriented verification (Section 5.4) research categories.
Simmonds et al. (2010a) extends previous ideas in the state-based conformity assessment (explained in Section 5.3), and proposes user-guided recovery framework which offers healing ritual for both safety and liveness violations. For the safety violations, a simple going-back action is performed, and for the liveness violations, both going-back and re-planning actions are carried out Simmonds et al. (2010b).

An infrastructure for system reconfiguration and failure recovery is presented in Mosincat and Binder (2008). Execution-manipulating self-healing ritual like service re-execution, isolation, replacement, or re-composition is designated in Wang et al. (2009). In addition, a detailed taxonomy on functional and non-functional fault categories as well as the corresponding healing strategies are also defined.

Generic QoS-driven web services monitoring framework and architectural-level healing ritual are represented in Halima R.B. (2007) that is implemented and evaluated in the context of European WS-DIAMOND (stands for Web Services DIAgnosability Monitoring and Analysis) project Console et al. (2008). The monitoring process and the healing ritual both are conducted at the level of SOAP messages based on the software layers to detect, analyse and repair QoS related faults Ben Halima et al. (2008).

Carzaniga et al. (2008) applies the notion of automatic workaround to the service-oriented applications as the modular systems providing redundant services that is required for the workaround process and healing ritual. In fact, a healing web service proxy mediates interactions between the service consumer and the service provider.

The ultimate edition of a self-healing system is to designate autonomous processes for not only recuperating the system from requirements deviations, but also adapting it to the coming new requirements and other unexpected situations. A self-adaptive distributed system is a closed-loop system which generally minimises the human-intervening activities that are regulated for the maintenance purposes. Undertaken studies in developing self-adaptive SOS are still in the early stages (Cheng et al., 2009).

MASC middleware Erradi et al. (2006b) is extended in Erradi et al. (2006a) to include certain adaptation processes for the web service composition. A dynamic reconfiguration framework is introduced in Paul and Tsai (2004) and Tsai et al. (2004) based on the new specification method that includes scenarios according to the ACDATE (Actors, Conditions, Data, Actions, Timing, and Events) model, and non-functional system constraints enforcement (e.g. security, performance, and reliability.). This approach is later applied in a Command and Control (C2) system that uses Policy Specification and Execution Language (PSEL) (Paul and Tsai, 2004).

Dynamic Service-Oriented Architecture (DySOA) presented in Siljee et al. (2005) deeply ponders self-adaptation from an architectural perspective. The intended purpose is to keep SOS adapted to QoS requirements that may be specified by a formal contract like SLA (see Section 3.4). An automated method for monitoring and dynamically reconfiguring workflows that is supported by the Web Service Requirements and Reactions POLICY language (WS-Re2Policy), and the corresponding adaptation process is described in Repp et al. (2008).

6.2 Prospective

Runtime verification process and self-healing and self-adaptation mechanisms are three succeeding facilities creating more reliable, fault-tolerant and dependable SOS with a capacity for adapting to stakeholders’ new requirements, changing business objectives,
market unexpected situations and stochastic chaotic operating environment. SOS still demands quite extensive research efforts on not only addressing various challenges in the literature, but also articulating and classifying many open problems in the context to direct scientific communities into these three divisions.

Acknowledgements

The authors are grateful to University of Kashan for supporting this work by Grant No. (92/6188).

References


Runtime verification of service-oriented systems


Note

1 By the term conformity we mean matching two state-based models.