ULRR

An analysis of formal languages for dynamic adaptation

Item Type	Meetings and Proceedings
Authors	Fox, Jorge;Clarke, Siobhán
Citation	15th IEEE International ConferenceEngineering of Complex Computer Systems (ICECCS), 2010;pp. 3 - 13
Publisher	IEEE Computer Society
Download date	2026-04-15 13:07:56
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/693

An Analysis of Formal Languages for Dynamic Adaptation

Jorge Fox
Lero - The Irish Software Engineering Research Centre
Distributed Systems Group
School of Computer Science and Statistics
Trinity College Dublin, Ireland
Email: Jorge.Fox@cs.tcd.ie

Siobhán Clarke
Lero - The Irish Software Engineering Research Centre
Distributed Systems Group
School of Computer Science and Statistics
Trinity College Dublin, Ireland
Email: Siobhan.Clarke@cs.tcd.ie

***Abstract*—The service-oriented computing paradigm is in widespread use for adaptive systems that face changing conditions in their operational environment as well as the integration of new services. In many domains, adaptations may occur dynamically and in real-time, using services from heterogeneous, possibly unknown sources. This motivates a need to ensure the correct behaviour of the adapted system, and its continuing compliance to time bounds. The complexity of dynamic adaptation (DA) is significant, but unfortunately currently not well understood or formally specified. Formal methods are an attractive option for solving this problem as they provide a means to precisely model a software system. There are many formal languages targeted to different domains, and in this paper, we present the results of our analysis of three languages as potential candidates for modelling our time-constrained DA problem. In particular, we selected JOLIE, PiDuce and COWS for analysis, as they are targeted towards service-based systems and each provide means to model at least some of our requirements. Our results illustrate the strengths and limitations of each, and justify our selection of COWS as the best-fit, though limited, language for our purposes.**

I. Introduction

Software adaptation supports modification of existing services in programs or inclusion of new ones, in response to inputs from the operating environment. Triggers for adaptation include changes in the running environment, availability of new services and reconfiguration issues. When dynamic adaptation occurs at runtime, it has to be performed within given time bounds and the resulting system must comply with the execution time established for the system.

Restrictions and requirements are imposed on the adaptation itself. For instance, when an existing service is substituted we need to identify the correct point in time to deactivate the former service. Substituting a service would usually imply severing its connections to the rest of the system and connecting the new service to the system. This involves, for instance, assuring that service interfaces are compatible and that the behaviour of the service to be replaced and the new one are equivalent in terms of system behaviour. Moreover, program monitoring is required to guide changes in the software system. For example, a decision-making process is needed to determine the moment when the process of adaptation has to occur. Formal languages provide the formal underpinnings to explain and model software systems in a precise manner. Even more, formal methods are fundamental for analysing software systems, validating and proving compliance to specifications. For these reasons we advocate the use of formal methods to achieve a model of DA. Problems associated with the modelling of DA can be better understood by first providing a formal model of DA and then exercising its practical application on particular software systems.

Selection of a best-fit language for DA is a non-trivial task, which remains an open question. In this work we endeavour to find a best-fit language for DA among some of the existing formal languages for services. This will support modelling of the underlying aspects of DA and provide a better understanding of it. Furthermore, this will provide an underpinning for DA systems. The importance of finding the right language to model DA lies in the need to be able to represent the main determinants of DA, such as composition, substitution, interaction, timeliness, and starting and killing processes.

In this paper we propose a set of specifications that represent our concepts of dynamic adaptation; based on these specifications we analyse selected formal service-oriented languages. This analysis is intended to provide a well-founded understanding of DA. We outline in Section II some requirements for service-oriented formal languages in DA which we use to explain strengths and limitations of the service-oriented formal languages we selected. The selection of languages was made based on a review of existing literature, looking for languages capable of expressing composition and interaction of services in view of dynamic adaptation. Therefore, we searched formal languages that represent dynamic changes of composites as well as their behavioural traits both internal and external with respect to the modularisation unit. This means we considered languages oriented towards services or components, specifically those representing interactions through channel-based communication. A relevant criteria was that the languages were able to describe changes effected on the composition of services or components allowing to modify the behaviour of the system in a precise, that is formal, manner. A short list of languages was first obtained and from this we chose the ones closer to the requirements we discuss in Section IV.

The remainder of the paper is organised as follows: We

978-0-7695-4015-3/10 $26.00 © 2010 IEEE
DOI 10.1109/ICECCS.2010.10

IEEE computer society

introduce the definitions that are relevant for achieving DA in Section II. These definitions are represented in the form of a group of requirements for the modelling of DA. Next, we introduce a case study used to explore the formal languages in Section III. Following this, we present an analysis of the three formal languages we selected and evaluate these in view of the requirements we established. Based on these definitions we identify strengths and limitations of these languages in Section IV. Further, we examine the advantages of each language and identify a best-fit language in the discussion of the results, and finalise with the conclusions.

In the following section we present the definitions and requirements.

II. Requirements on Languages for DA

The representation of DA demands language constructs capable of describing processes such as service substitution, service elimination, and changing service behaviour. Another important feature of DA is the decision making process relating to when and how the adaptation has to be performed. The decision making process can be carried out by an adaptation manager monitoring the whole system or by each service on its own. This means we may have two types of adaptation manager, a global one, and a service specific one.

Therefore, we require from a language that it be able to represent these concepts in a precise manner.

Aspects of DA relate mainly to the adaptation of services, which can be decomposed into upgrading a service, adding a new service and deactivating a service. In runtime DA the time factor plays a crucial role. Consider the time needed for adding a new service and the time bounds established for the correct execution of the system. Decisions on performing or aborting an adaptation are tied to time bounds. For that reason the language we choose for modelling DA has to provide the constructs to comprise time in the model.

The following definitions represent the core elements for DA which we identified after a thorough review of current work on adaptation [2], [3], [4], [5], especially on DA. The aim of this group of definitions is to help us assess the potential of selected (formal) service-oriented modelling languages.

Service adaptation can be represented by base services and the service(s) of the adaptation process. We define a base service and another service that replaces the base one as follows.

We define $s \in S$ as a service, where S is the set of services in the system.

Definition 1: A **service** is a computational entity that performs a given functionality and has no sub-services or components within itself. A service has the following elements:

An **interface** with input and output communication channels denoted *in* for input channels and *ou* for output channels. The interface I consists of both types of channels $I = \{in, ou\}$. Where *in* and *ou* indicate input or output channels respectively.

A service has a **location** $loc \in L$ where L is the set of all possible locations and a **name** $nam \in H$ where H is the set of all service names.

Service communications occur through **channels** that is the service interface and recipients of a message are addressed by *loc* and *nam*.

Dynamic adaptation in service-oriented systems requires a function to modify system behaviour as well as a function to deactivate selected services.

Definition 2: A **function to modify system behaviour** by upgrading a service is a function of the form:

$s_2 \rightarrow_{subst} s_1$

Where $\rightarrow_{subst}$ indicates a function that takes a service s_1 and substitutes it with another service s_2.

Definition 3: A **deactivation process** $deac(s)$ deactivates a service by sending a signal to the runtime environment, which in turn effects the deactivation.

Deactivation of a service can be achieved by erasing the service from the registry. In such a case, we must make sure that service s_1 has completed its execution before deactivating it.

Therefore, a boolean function is required to know if service s_1 is running at a given point in time. We define a function $running(s)$ that returns a boolean value, true if the service is currently running, false otherwise. The deactivation process and the substitution process have to wait until $running(s)$ returns false.

The actual deactivation of s_1 can be effected by a function $shutdown(s)$ provided by an adaptation manager, middleware, or runtime environment.

Definition 4: Substitution of s_1 with s_2. The **substitution process** has the form subst($s_1, s_2, tsubst_{max}, texec_{max}$). Where channel names of $s2$ are given those of s_1, $s_{2 \cdot ou} := s_{1 \cdot ou}$ and $s_{2 \cdot in} := s_{1 \cdot in}$. Timeliness is considered in the variables $tsubst_{max}, texec_{max}$.

Therefore, following Definition 4 s_2 needs to have its channel names and location changed to those of s_1. This means renaming s_2 channels and name.

Regarding services that interact with s_1, further changes are not required, since $subst(s_1, s_2)$ converts channels, location and name from s_1 to s_2. However, the process has to **synchronise the substitution** without affecting running processes, specially if s_1 is running when we want to substitute or deactivate it. To achieve this the adaptation process requires a channel rename function. A function to modify system behaviour, deactivation, and substitution functions are also required to model DA.

Definition 5: Channel rename function $s_{2 \cdot in} := s_{1 \cdot in}$ $s_{2 \cdot ou} := s_{1 \cdot ou}$

Where $s_{2 \cdot in} := s_{1 \cdot in}$ and $s_{2 \cdot ou} := s_{1 \cdot ou}$ assign the names of the channels of s_1 to s_2.

The service **name** *nam* in s_2 is changed into that of s_1 by assigning it as follows, $nam_{s_2} := nam_{s_1}$. The **location** is also modified in the same manner: $loc_{s_2} := loc_{s_1}$

s_1 and s_2 can not execute simultaneously. s_2 can only be started after deactivation of s_1.

There is no need to modify the services that interact with s_1, since s_1's location, name and channel names are assigned to s_2 (the new service).

Another requirement for the adaptation process is to care for correct and timely changes from s_1 to s_2. This means, the adaptation process has to guarantee that the adaptation time is within specified time bounds, also that the execution time of the new service is within defined time bounds and service behaviour complies with the system's specifications.

Having defined the main determinants of dynamic adaptation at the architectural level for service-oriented systems, we now move to reflect on the proposed mechanism to execute adaptations. This mechanism we denote as adaptation manager.

Where timing considerations have to be fulfilled, the adaptation manager (AM) receives time estimates on service substitution and execution time from the environment or related services.

Adaptation Manager

The AM substitutes s_1 with s_2, it also evaluates the execution time of the new service $timeexec(s_2)$ and the time it takes to substitute service s_1 with s_2, $timesubst(s_2)$. The AM also handles location and names of both services in order to connect the new service to the rest of the system.

The adaptation process requires some mechanism to direct the changes required to achieve DA in a given system. Current services in a system are usually designed to perform a particular functionality. The adaptation process should be performed by a third service that operates as AM. We propose an adaptation manager to effect adaptations since some is needed to control adaptations, for which a mechanism such as a service registry would not suffice. Even more, the AM performs a number of functions, handling adaptation requests, controlling timeliness aspects, and conduct service substitution.

The AM is concerned with timing issues. Timeliness can be modeled by adding a variable $t_{max} \in N^+$ in $subst(s_1, s_2, tsubst_{max}, texec_{max})$. Where $tsubst_{max}$ represents the maximum time span allowed for the substitution process to take place and $texec_{max}$ is the value of the estimated execution time of s_2. This is illustrated in Figure 1. The AM provides time estimates on service substitution and execution time. Subsequently, we propose the substitution and deactivation steps in Figure 4 that illustrate the incorporation of timing considerations in the adaptation process.

if $(timeexec(s_2) \leq t_{max} \leq texec_{max})$ and
$(timesubst(s_2) \leq tsubst_{max})$
then <begin>
// rename channels
$s2._{in} = s1._{in}$
$s2._{ou} = s1._{ou}$
// rename loc and name
$nam_{s_2} = nam_{s1}$
$loc_{s_2} = loc_{s1}$
// commit changes
$deac(s_1)$
$commit(s_2)$
<end>
else abort
Where $deac(s_1)$ is a deactivation routine that ends the execution of service s_1, the one being substituted and $commit(s_2)$ indicates the completion of the substitution by launching service s_2.

Figure 1. Timeliness in the adaptation process

III. Tollbooth Scenario

This section introduces the scenario in which we apply the definitions presented in the previous section. The scenario will help to illustrate some aspects of the languages in view of the definitions. The scenario is about a tollbooth system. The flow of events is given below. In this scenario a car approaches a tollbooth and gets a welcoming signal from it with information on the protocol required to pay at the tollbooth. Afterwards, the car verifies the protocol and if needed adapts its electronic toll system to comply with the tollbooth's protocol. The scenario follows. We relate the requirements previously introduced to the steps in the tollbooth scenario in Table I.

A. The vehicle approaches the toll area through the circulation lane.
B. The vehicle reaches the physical domain of the tollbooth.
C. The vehicle's electronic toll system (ETS) receives a welcoming signal from the tollbooth system.
D. The tollbooth system requires the vehicle's ETS to confirm compatibility and version of ETS.
E. The ETS protocol has to be the same as the one in the current tollbooth area.
F. If the protocol is not the same, a new one is required in the form of a service.
   a) The tollbooth system sends a service (with the protocol) to the vehicle's ETS.
   b) The vehicle must run a service adaptation procedure to substitute the previous service with the new one.
   c) The tollbooth estimates the time the vehicle requires to arrive to the actual tollbooth. This time estimate is used as the time limit for the adaptation process.
   d) Vehicle's AM estimates the adaptation time. If estimated adaptation time $\leq$ time of arrival to tollbooth, then it proceeds to evaluate execution time. Otherwise a signal is sent to the driver, so he moves to a cash payment tollbooth lane.
   e) The AM estimates the execution time of the new service. In case it complies with current time bounds of the system, the adaptation proceeds, otherwise it is aborted and a signal is sent to the driver, so he moves to a cash payment tollbooth lane.
   f) In case of successful adaptation, the vehicle's ETS notifies the tollbooth about the installation of the new service.

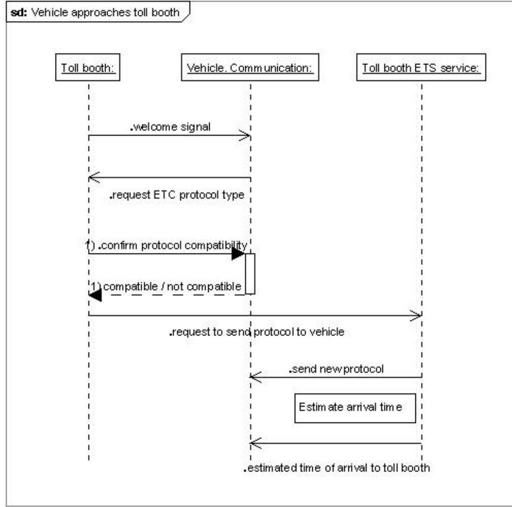

Figure 2. Scenario. Sequence diagram Vehicle approaches tollbooth

g) The tollbooth system debits the toll from the vehicle's money pocket.
h) The driver approaches the tollbooth and drives through.

Requirement	Step
Service substitution	F-(b)
Service elimination	F-(e)
Service adaptation	F-(b)
Time bound adaptation	F-(d)
Time bound execution	F-(e)
Decision making (AM)	F-(d), F-(e)

TABLE I
RELATION OF REQUIREMENTS TO STEPS IN THE TOLLBOOTH SCENARIO

The ETS has access to an electronic money pocket that is adjusted to a given tollbooth protocol, when the protocol is available.

Modifying the system to comply with a new ETS protocol, steps F-a and F-b, in the tollbooth scenario, have to be performed at runtime. The adaptation is executed by a service that substitutes the former one. This demands DA, since it has to be performed at runtime.

A graphic representation of the scenario and adaptation process can be found in Figures 2 and 3.

IV. ADEQUACY OF (FORMAL) SERVICE-ORIENTED LANGUAGES TO MODEL DA

The goal of this section is to describe limitations and possibilities of the selected service-oriented formal languages. We analyse capabilities and limitations of three formal languages in view of DA, namely SOCK/JOLIE, PiDuce and COWS.

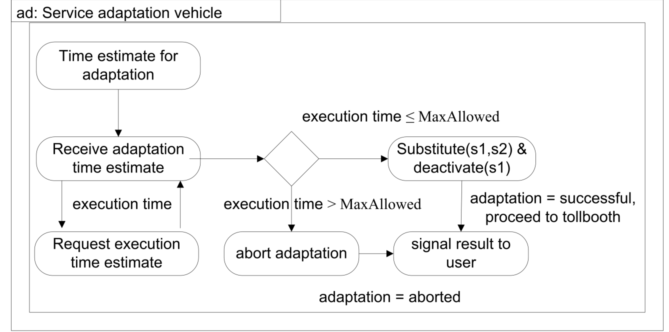

Figure 3. Scenario. Service adaptation procedure

A. Analysis of SOCK/JOLIE for DA

The Service Oriented Computing Kernel (SOCK) is a calculus that formalises central concepts of service oriented computing. These concepts include design of service behaviour, service deployment and composition within a system. In SOCK service design is decomposed in three parts: behaviour, declaration, and composition [6]. SOCK defines the basic mechanisms for these concepts. It is divided into three layers: service behaviour, service engine and services system [7]. Accordingly, it is formed by three calculi: service behaviour calculus, service engine calculus, and services systems calculus. Service behaviour calculus is used to design service behaviours by means of computation and external communication primitives, inspired on Web Services operations and workflow constructs such as sequence, parallel and choice. The second calculus serves to specify execution modality, persistent state flag, and correlation sets. The third calculus, services system calculus, defines the whole system including interactions among services [6]. SOCK's layering separates the different aspects of service oriented computing in order to explore them separately as desired. We quote from the work of Guidi et al. [6], [8], [9] to explain this formal language in the following lines.

The semantics of the calculi are defined with labelled transition systems, all arranged in five layers: behaviour, service engine state, service engine correlation, service engine execution modality, and services system.

The service behaviour calculus exploits external input and output actions for communication with other services. Communication is effected by means of operations that resemble those of Web Services. Operations are named and have an interaction modality. There are four kinds of peer-to-peer interaction modalities divided in two groups: Input and Output operations. Input operations can be One-Way or Request-Response, while Output operations are Notification or Solicit-Response. Furthermore, operations are grouped into single message or double message operations. One-Way and Notification are single message, while Request-Response and Solicit-Response are double message operations.

The service behaviour layer models the internal behaviour of the service and communication primitives. The service engine

layer represents the execution environment of the service. The composition of service engines is outlined at the services system layer. Single message operations names are represented by the set $\mathfrak{O}$ and double message operations names by the set $\mathfrak{O}_R$, these sets are disjoint. Consequently, the set $S_{UP} = \{(o, ow)|o \in \mathfrak{O}\} \cup \{(o_r, rr)|O_r \in \mathfrak{O}_R\}$ is the set containing all the input operations, where o_r and rr are One-Way and Request-Response operations. The set of all output operations is described by $Inv = \{(o,n)|o \in \mathfrak{O}\} \cup \{(o_r, sr)|O_r \in \mathfrak{O}_R\}$, where n and sr represent Notification and Solicit-Response operations respectively. All the possible operations are indicated by the set $Op = S_{up} \cup Inv$. The set of locations names is denoted by Loc, ranged over by l. The service engine calculus is made of the concepts of *state*, *correlation sets*, and *service declaration*. Conditions can be evaluated over states. This facilitates checking conditions on the execution of services. Engines have a location that identifies them. The calculus syntax is: $E ::= Y_l \mid E \mid\mid E$. Where a service engine E con be a located engine Y_l or a parallel composition of engines. The syntax and semantics of SOCK enable a formal analysis of service behaviour. This is one reason to consider SOCK in our analysis of formal language for services. For a more detailed description of SOCK the reader may refer to Guidi et al. [6], [8].

The implementation of the formal calculus SOCK is JOLIE. It is a Java interpreter for a subset of the orchestration language SOCK. In it, services communicate with other services by exchanging external input and output actions. Such internal and external actions are called operations and are defined through a name and an interaction modality. SOCK proposes four kinds of peer-to-peer interaction same as the interaction modalities in the service behaviour calculus of SOCK, these can be classified in two groups: *input operations* and *output operations*.

One characteristic of this language is that input operations are published by the corresponding service. Output operations are used for sending messages to the input operations of other services. Therefore, either input or output operations have to be published. Specifically, input operations are defined by an *InputPort* and output operations by an *OutputPort*.

Following from the service behaviour calculus, messages in JOLIE are classified into single and double message operations, in accordance with the definitions of SOCK. Single message operations are called *One-way* and *Notification messages*. Double message operations are called *Request-Response* and *Solicit-Response* operations.

Locations for input and output ports are defined at design time as opposed to a dynamical allocation of addresses. In summary, in order to achieve a correct message delivery in JOLIE both location and operation name are needed, and locations are determined at design time. This can be considered a limitation in view of DA.

Services in JOLIE have input and output ports used to communicate and interact with other services in the system. Ports are akin to interfaces, in this regard this language complies with Definition 1. In addition, ports in JOLIE are

Requirement	Step in toll booth scenario	Result
Service substitution	F-(b)	Dynamic embedding, redirection, and aggregation
Service adaptation	tb:messaging and tb:ETS service try to connect to v:communication	Above traits plus only the first message may start
Time bound adaptation	F-(d)	No support
Time bound execution	F-(e)	No support

TABLE II
COMPLIANCE OF SOCK/JOLIE TO THE REQUIREMENTS

assigned a location i.e., address, which is also consistent with Definition 1. However, JOLIE provides no mechanism or process to modify system behaviour (see Definition 2) and no deactivation process (see Definition 3).

Given that channel names can be modified only at design time, as opposed to runtime, JOLIE does not comply with Definitions 4 and 5.

Service interfaces and locations are also defined at design time and can not be modified at runtime. Consider step F-b in the tollbooth scenario, an adaptation procedure has to run in order to substitute a service with a new one, however in SOCK/JOLIE the substitution of a service by another one is not possible since the language focuses on orchestration of services and is not designed to allow for service substitution. Such a limitation also points to the non compliance of the language with Definition 2.

In our case study, the *ETS service* sends a new protocol to the vehicle's *communication* service, the subsequent action in the vehicle would be to add the new protocol or substitute the existing one, but this can not be modelled by SOCK/JOLIE, since there is no substitution function or mechanism.

Moreover, message links between services are established at design time and can not be modified at runtime except for the case of failure handling [10], which performs a limited kind of adaptation.

As another example, consider service interactions. We identified two cases for service interaction. In the first case, (Figure 2) two services like *Vehicle communication* and *Tollbooth ETS service* try to connect to an input port of service *Vehicle communication*, this is a port that has a unique location. As a result, the first service may start, but the second service gets an error message: "Address already in use: bind" and halts. Therefore adding an additional service to *Vehicle communication* requires defining a new port for it. This has to be performed at design time.

In the second case, two services address the same output port of a third one. There are no address bind problems. However, JOLIE allows different services to direct messages to a given output port from other services. For instance, consider the communication exchange between the communication ser-

vice (marked Vehicle communication) and the *ETS* service in Figure 2. Particularly, sending a new protocol to the vehicle may require some mechanism to substitute the existing service that has the older *ETS* protocol and substitute it with the new one. This requires to make changes in service addresses and locations as stated in definition 4, JOLIE does not support changing service addresses as shown in Table II.

Additionally, communication between services is accomplished via output and input ports that are predefined in the services themselves, therefore other services have to be aware of these ports and locations in order to interact.

In relation to our case study, the addition of a service such as the new protocol implies defining new input and output ports.

According to Guidi and Lucchi [11], orchestration languages, such as JOLIE, support service composition. A limited form of DA can be achieved, as in the case of failure handling. Guidi et al. [10] propose that fault, termination and compensation handlers do not necessarily need to be statically defined, but can be updated at runtime. For this ends, they propose a *dynamic handler installation*. We quote their following explanation on scopes:

> Technically, we consider a scope construct of the form scope_q(P,H) where q is the name of the scope, P is the process to be executed, and H is a function associating fault handlers to fault names and termination and compensation handlers to scope names.

Where a scope is "a process container denoted by a unique name and able to manage faults"[10].

To achieve dynamic fault handling the authors developed a set of primitives. Moreover, the previously installed handler implies that "the language must be able to generate behavioural code dynamically" [12]. The code is dynamically generated by the interpreter. Fault handler installation is done via a specifically designed primitive, rendering this solution to be particularly tailored as opposed to a more generic one. Another limitation is that the fault handler has to be installed at design time [12]. DA in JOLIE depends on having the handling and installation primitives previously defined and the dynamic fault handlers "hard coded". This differs from our concept of DA in which we require mechanisms that add behaviour at execution time without previous hard coding of the adaptation handlers.

As illustrated in Table II, JOLIE in its present form, provides limited support for DA. The extent to which JOLIE supports dynamic adaptation is primarily focused on the definition of failure handlers at design time and the dynamic generation of code at runtime for these handlers. Nevertheless, JOLIE supports another mechanism for DA by means of containers, where a container is an application able to execute more than one service. This provides enhanced possibilities for service composition and dynamic service composition within containers, as well as redirection and forwarding of messages from a master service to multiple output ports. In this way, SOCK/JOLIE facilitates dynamic embedding, redirecting and aggregation of services [8].

We consider that JOLIE requires mechanisms or processes to attain and control modification of system behaviour (see Definition 2), a deactivation process (see Definition 3) as well as resetting and renaming of channels in order to improve its capabilities in view of DA. One drawback we identify is that there is no support for modelling timeliness in this language.

B. Analysis of the PiDuce Distributed Machine

PiDuce is part of a project for experimenting with (Web) services technologies. This language belongs to the domain of "distributed abstract machines for pi-like calculi" [14]. Programs in the distributed pi-calculus interact with other programs via channels and message passing. Message communication through channels is common in several formal modelling approaches.

PiDuce was designed to implement a distributed π-calculus and to provide a distributed machine for Web services [14]. The machine for Web Services consists of a prototype, a runtime environment and a Web interface. The PiDuce machine is able to communicate with remote services (PiDuce services). Since π-calculus relies on name passing and a service is equated to a π-calculus channel, PiDuce may model Web services.

In PiDuce, processes and channels are static while at the same time messages travel through the network. Services are defined as channels. Channels are "first-class" citizens in the language. They are equivalent to values and can be sent and received from other channels [14].

The underlying model of PiDuce is distributed and consists of a number of runtime environments. These runtime environments are made of a virtual machine and a channel manager.

In this section we analyse PiDuce in view of the requirements for DA that we set in Section II. We outline a parallel between the definitions we introduced in that section to the published work of PiDuce [14]. Next, we select examples from the available information on the language and introduce our examples to explain the extent to which PiDuce supports DA.

This language is built on the π-calculus. A program in this language uses channels and message passing to collaborate with other programs. Services are defined as channels. This differs with Definition 1 which is a more service or component-oriented one, while the definition of service in PiDuce is rather a logical one. This language apparently diverges from a service-oriented approach with respect to the definition of services as we introduced it. Services in PiDuce have a location and this complies with Definition 1. For instance, in scenario *vehicle approaching tollbooth* Figure 2 there are input and output channels between services *tb:messaging* and *v:communication*. A message such as *Welcome signal* is sent through channel *tb:messaging* to channel *v:communication*. This can be modelled in PiDuce as channels and message passing, where the channel is the connection of *tb:messaging* with *v:communication*, and the message is the *Welcome signal*.

The basic model for distributed systems in the asynchronous π-calculus assigns channels to a single location. This raises the

Requirement	Step in toll booth scenario	Result
Service substitution	F-(b)	Channel renaming
Service adaptation	tb:messaging and tb:ETS service try to connect to v:communication	Adaptation through channel renaming
Time bound adaptation	F-(d)	No support
Time bound execution	F-(e)	No support

TABLE III
COMPLIANCE OF PIDUCE TO THE REQUIREMENTS

problem of "input capability." Input capability "is the ability in the (asynchronous) π-calculus to receive a channel name and subsequently accept inputs on it" [14].

The following example illustrates input capability.

> Consider the example from S. Carpineti et al. [14], *x(u).u(v).Q*. This program is located at (the location of) *x*, but upon reaction with $\bar{x}$*[w]* it produces the continuation $w(v).Q\{w/u\}$ and this continuation is still at x, whereas it should be at *w*.

In our case study this can be exemplified by the messages *confirm ETS's compatibility* and *send new protocol* which are sent to *v:communication*, but in the calculus the continuation of the messages would still be at the sending service not at the receiving one.

As explained by Carpineti et al. [14], in a term like *x(u).P* the process P can not accommodate any inputs on *u*. This problem is solved by using "linear forwarders". A linear forwarder takes one message from channel *x* and sends it to channel *y*. Linear forwarders deviate the message to another channel and the system may then handle more messages. This is equivalent to a channel rename function as defined in Definition 5. This way, PiDuce provides some support for adapting services through renaming of their channels. One aspect that remains open is the compatibility of service interfaces, which has to be considered when modifying the connection of a service in the system. Service names can also be generated dynamically.

An interesting feature of PiDuce is that channels are values which may be sent over and received from other channels. This feature may support sending and receiving services, conceived as channels, for dynamic adaptation. For example, consider the services *tb:ETS service* and *v:communication* in Figure 2. The first service needs to send a new protocol to the communication service, PiDuce can achieve this by sending the new protocol as a channel to the communication service.

PiDuce offers support for creating and substituting channels, however processes themselves can not be modified or substituted, as illustrated in Table III. Processes are defined statically and adaptation of services can be made only through channel creation or substitution. The capabilities of PiDuce for dynamic adaptation are limited by the static service structure as well as the lack of an explicit adaptation mechanism. This language does not support modelling of timeliness.

C. Analysis of the Calculus for Orchestration of Web Services (COWS) for DA

The calculus for orchestration of web services (COWS) [15] is a new process calculus similar to the Business Process Execution Language (WS-BPEL) [1]. However, contrary to WS-BPEL, COWS provides a formally specified distributed machine. COWS is a process calculus that provides for the specification of service-oriented applications, as well as for modelling dynamic behaviour. COWS is also intended to provide support for the development of tools that allow to check that a given service composition follows desirable correctness properties and unexpected behaviours are avoided.

COWS has been extended with timing elements [16] which facilitate adoption of the language for modelling services with timing requirements. Therefore, this language represents an important line of research in service-oriented computing (SOC). The motivation of the authors to develop COWS is that most formalisms "do not model the different aspects of currently available SOC technologies in their completeness" [16].

In COWS services are structured activities built from basic activities, such as the empty, kill, invoke, receive, and wait. Services are composed by means of prefixing, choice, and parallel composition. Constructs as protection, delimitation, and replication are also provided. The language is parameterised by a set of expressions. Partner names and operation names can be combined to designate communication endpoints, written "$p \bullet o$", where p is a partner and o an operation. These represent activities of type receive "$p \bullet o?$" or invoke $p \bullet o!$".

Endpoints are exemplified in Figures 4 and 5 by the partners *s, amadapt*, and *aminstall* and the operations *amadaptOK, launchOK, adaptreq*, and *launchFail*. Their interaction is indicated by the signs ? and ! as receive or invoke respectively.

Timed activities are frequently exploited in SOC and are needed to model time outs. Passing of time is modelled synchronously for services deployed on a same 'service engine', and asynchronously otherwise. In this language, time passes synchronously for all services in parallel. Services run on a same service engine. An important aspect in dynamic adaptation is represented by timing constraints for either the execution of a new service or the execution of the system as a whole, therefore elements for modelling time are needed. Timing considerations are absent in the previous two approaches.

In the tollbooth steps F-d and F-e can be modelled as timed activities and the time bounds are given by the time estimated for arrival to the booth.

A short example at the case of the tollbooth can be modelled as a predicate that awaits for an adaptation and aborts if it exceeds a given time, say 5 units of time. The predicate would look as follows.

adaptation manager = $*[x_B, x_{id}] \ \ldots \oplus \bullet_{abort} \mathrm{x}_{id}!$

Which means that if the process exceeds 5 time units it is aborted.

Requirement	Step in toll booth scenario	Result
Service substitution	F-(b)	Conditional choice and sequential composition
Service adaptation	F-(b)	Conditional choice and sequential composition
Time bound adaptation	F-(d)	Passing of time modelled by explicit actions
Time bound execution	F-(e)	Passing of time modelled by explicit actions

TABLE IV
COMPLIANCE OF COWS TO THE REQUIREMENTS

```
let [service1=service to replace,
service2=new service]
aminstallrequest = s.amadapt?<service1,service2,
loc1,loc2,adaptTime,adaptDeadline>.
(s.aminstall!<service1,service2,loc1,loc2,
adaptTime,adaptDeadline>
| s.kill!<service1>)

s.aminstall?<service2,loc1,loc2,adaptTime,
adaptDeadline>.install!<>
s.install?<>.
(if (timeexec(s2) <= adaptDeadline) and
(timesubst(s2) <= adaptTime)
then rename channels
loc2.input = loc1.input
loc2.output = loc1.output
launch(s2)
deactivate(service1)
else ( abort | s.launchFail!<> ))

requestor =
amadapt.aminstall!<service1,
service2,loc1,loc2,adaptTime,adaptDeadline>

signalrequestor =
s.launchOk?<sname>.s.signalOk!<>
```

Figure 4. AM for COWS with timeliness aspects

COWS [17] provides a proxy mechanism to add compensation handling in existing services. This means that it may add behaviour to existing services in the system by appending a compensatory service. A compensatory service is one that adds to the base service and achieves some corrective or compensatory functionality. In our scenario, this means that we add a protocol through a compensatory service. We can also avail of sequential composition for adapting an existing protocol to the new one.

COWS can model several typical aspects of (web) services technologies, for instance, multiple start activities, receive conflicts, routing of correlated messages, service instances and interactions among them [16].

For instance, consider the illustration in Figure 3. The execution time estimation is sent to the adaptation manager and from this one to the substitute and deactivate services. The process is enacted when the execution time is within the predefined time bounds, otherwise the adaptation process is aborted. COWS provides the framework to cater for this type of process.

In COWS basic actions are *durationless* and the passing of time is modelled by explicit actions.

Imperative and orchestration constructs support specification of assignment of variables, conditional choice and sequential composition. Conditional choice and sequential composition of services can be used to attain dynamic adaptation by composing services with existing ones. The language defines further the concept of service engines, where each engine has its own clock which is synchronised with the clock of other parallel engines. All instances of a service run within the same engine. Moreover time elapses between each evaluation of expressions and these evaluations are instantaneous. Only the time construct $\oplus_{argument}$ consumes time units. Time elapses while waiting for invoke or receive activities and the argument of wait activities "$\oplus$" is set to the current stand. Parallel composition of engines in this language is given by sequential composition, which complies with Definition 2. Regarding Definition 3, COWS provides a deactivation activity that forces termination of all unprotected parallel activities, and sensitive code can be protected from killing by placing it into a "protection". This characteristics are represented in Table IV.

COWS computational entities are called *services* [15]. Services in COWS do not have interfaces since communication is realised through message passing among services, which are structured activities built from basic activities. Relating COWS to the requirements in Section II is not straightforward since this language has no component model, but provides high level abstractions to model software systems and this is the reason to consider it a good option for modelling DA, specially taking into account that it has explicit timing constraints which the other languages do not provide.

COWS does not have an adaptation manager. It would therefore require the definition of explicit constructs to allow it to direct the adaptation process and perform the activities of an adaptation manager, as defined in Section II. The capabilities of COWS to model dynamic adaptation would also be enhanced by adding primitives or mechanisms for request-response signals. This way the adaptation process can be achieved by an adaptation manager that sends an adaptation request and awaits for a confirmation in order to kill the original service once the new service has been enacted. This adaptation manager can be outlined as a service effecting the steps we propose in the adaptation process introduced in Figure 1 where the service to be replaced and the new service would be referred to by their names and locations.

```
let am(sname,repsvc,deadline,s,r) =
am.adaptreq?<sname,repsvc,deadline>
| amcheck.adaptime!<deadline,s,r>
| adaptreq.amcheckOK?<>.requestor.
amOK!<sname,repsvc>
+ adaptreq.checkFail?<>.requestor.
amFail!<sname,repsvc>
amcheck(sname,deadline,s,r) = amcheck.tTotal?<sum>
|amcheck.adaptime?<deadline,s,r>
|amcheck.amFail!<>
amadapt(sname,repsvc,deadline,s,r) =
amadapt.amOK?<sname,repsvc>.amadapt.
inst!<repsvc>
|amadapt. adaptreq?<sname,repsvc,deadline,s,r>.
amcheck.tTotal!<s,r>
|amadapt.amadaptOK!<> | aminstall.
launch!<sname,repsvc>
|amadapt.amadaptFail?<>.amadapt.
okshutdown!<repsvc> |amadapt.failshutdown!<repsvc>
aminstall(repsvc) =
s.launch?<>.aminstall.
launchOk!<sname,repsvc>
```

Figure 5. AM for Cows without timeliness aspects

COWS also requires primitives to model state, which in the case of adaptation would grant the possibility of transferring current state of the service to substitute to the new one and continue execution without interruption of the whole service. We outline a proposal for an adaptation manager that checks for timeliness constraints in Figure 4 and another version without timeliness constraints in Figure 5, keeping in mind that the former does not represent an actual implementation in this language and some language elements would need to be specifically designed and added to it. In this proposed code we consider a call to a service "aminstall" that receives from an install requestor "aminstallrequest" the old service and the new one together with the values with the time it takes for adapting and the maximum time permitted to adapt, this service verifies the time needed to adapt with respect to the maximum time allowed to perform the adaptation process if this time is not exceeded then the input and output channels of the new service are related to those of the replaced one. Then the new service is launched and the replaced one is deactivated. Finally, a signal is sent to the User to indicate success or failure of the process. The execution time of the new process is not considered in this version, which would have to be included in "aminstall" in order to comply with the required execution time of the system after adaptation.

V. Other Formal Languages for DA

A. KLAIM

KLAIM is a language that provides constructs for global computing. This language is based on process algebra and uses a coordination-oriented approach to systems design. This language was proposed to design distributed systems made of several mobile components. These components communicate through tuple spaces. Localities are treated as first-class elements and can be dynamically created and communicated. Communication is effected via distributed repositories and remote operations [18], [19]. DA in KLAIM can therefore be modelled as mobile components being dynamically created and their communication links updated accordingly. One limitation we identify is that KLAIM does not provide constructs to evaluate timeliness.

B. Service Centered Calculus (SCC)

SCC is a process calculus that provides precise notions of service definition, service invocation, and bi-directional sessioning. It is inspired by the π-calculus. Interaction modalities are modelled by explicit sessions on the client and the server side. Even more, SCC allows for closing of sessions, which KLAIM does not offer. In SCC services are regarded as interacting functions or stream processing functions invocable by clients [20]. SCC is a name passing process calculus that allows to create and invoke services. In SCC service invocations produce new sessions. This is the way SCC models interactions between clients and services. It is not clear to what extent we may model DA based on SCC's constructs, since these are more oriented towards modelling service interactions rather than composition or modification.

VI. Discussion

After exploring the main aspects of the selected formal service-oriented languages, we need to select a best-fit language for DA out of the three languages studied. We believe a valuable contribution for the further understanding of its challenges and a formal language will allow us to develop precise and verifiable models of DA. Piduce is a machine that is able to communicate with remote services, however processes and channels are static. It does not provide a mechanism or process to modify service behaviour and no deactivation process. These two properties are essential to DA. Moreover, channel names can only be modified at design time not at runtime. However, it provides a limited adaptation of services by means of channel renaming through a linear forwarder. An advantage of PiDuce is that service names can be generated dynamically and creation and substitution of services is supported. Yet services and locations are defined at design stage and can not be modified at runtime. A further drawback of PiDuce is that in order to achieve correct message delivery both location and operation of the service are required. SOCK/JOLIE introduces a service layering that facilitates separating the different aspects of service-oriented computing. SOCK is the formal language and JOLIE is the interpreter. Message links are established at design time and can not be modified at runtime except for the case of failure handling, which performs a limited kind of adaptation. It relies on predefined constructs for code generation and service interaction particularly tailored to a given problem. Furthermore, service addresses can not be changed, which limits its flexibility in view of DA. Equally important,

output and input ports are defined at design time and can not change at runtime. SOCK/JOLIE is therefore not a feasible alternative to model DA. COWS supports the development of tools for checking correctness properties and avoidance of unexpected behaviours. It also has timing elements, which make it suitable for runtime DA under timing constraints. Timed activities are frequently exploited in service-oriented computing and are used to analyse time outs. This language has imperative and orchestration constructs through which conditional and sequential composition can be represented. It also has a deactivation activity. Adding behaviour can be achieved through compensation handling in existing services. This way, it is possible to add behaviour to existing services by appending a compensation service. A limitation is that there is no component model, but there are high level abstractions to model software systems. There is no adaptation manager and therefore the definition of a construct or process is required in order to be fully adequate for DA. A main difference between COWS and SOCK, as identified in [21], is that COWS is stateless and bases correlation on values, while SOCK is stateful and bases correlation on variables. Moreover, SOCK allows to change correlation information dynamically, while COWS does not. This is a limitation of COWS to fully achieve DA. Some limitations of KLAIM and SCC in view of DA have already been outlined in Section V-A and Section V-B. We included these two languages to enrich the spectrum of languages considered, yet some criteria from our case study was not evaluated with these due to the more theoretical information provided for these two languages, which was much more oriented to relating them to congruence analysis and equivalence of with π-calculus and process algebra rather than their application.

We conclude that the best-fit language for runtime dynamic adaptation is COWS. Given the characteristics of the languages selected, where only one provides constructs for timing analysis, it is no surprise that our choice as best-fit language is precisely this one, COWS. However, it will need to be enhanced by adding an adaptation manager, which can be developed using the language itself. However, timeliness was not the only deciding criteria, consider to what extent adaptation of services merely via channel renaming is sufficient to achieve DA is questionable, as is the case of PiDUce. In this regards the composition mechanisms of SOCK/JOLIE are more adequate, yet again with no possibility to evaluate timeliness. Our choice is clear and well founded.

VII. Related Work

Dynamic Adaptation

We find an overview of DA and its constituents in the work of Mckinley et al. ([4], [5], however they do not advance a formal model or proposal to explore DA, which is the aims of our work. Similarly to the elements of DA we identified, Segarra and André describe a similar model to ours with components that can be customized for different applications, a component in their framework can be provided with a controller which performs the adaptation depending on execution conditions [23]. In our proposal we define one controller, the adaptation manager, that gathers information from supporting services such as timing and execution evaluation in order to perform adaptations. The Service Centered Calculus (SCC) [20] features explicit notions of service definition, service invocation and session handling. This language is basically directed at orchestration of services. The language does not provide operators for explicit closing of sessions, which is required to model dynamic adaptation. Since it is aimed at orchestration we did not consider it in our selection of service-oriented languages. WS-BPEL aims at providing a language for the formal specification of business processes and business interaction protocols. Its focus is on service integration rather than service adaptation. Work has also been carried out to map BPEL to Process Algebras as Ferrara [24], to Pi-calculus as Abouzaid [25], and to Petri Nets as Ouyang et al. [26].

Formal approaches to DA

The work of Laneve and Zavattaro [27] on web services advances an extension to the π-calculus with a transaction construct, the calculus webπ. This model supports time and asynchrony. However it remains at a more abstract level and is not applied to dynamic adaptation. Ferrara [24] relies on process algebra to design and verify web services, this work also allows to verify temporal logic properties as well as behavioural equivalence between services. Compared to this work, our attempt is more general and is directed at the study of dynamic adaptation. Finally our proposal is aimed at identifying a formal service-oriented language for modelling dynamic adaptation, rather than advancing techniques for formal verification of web services or services as in the work of Ferrara. Mori and Kita [28] explore the use of genetic algorithms to dynamic environments and offer a survey on problems of adaptation to dynamic environments. The work of ter Beek at al. [21] reviews service composition approaches with respect to a selection of service composition characteristics and helps to underscore the value of formal methods for service analysis at design, specially service composition. The authors present a valuable analysis of formal approaches to service composition and elaborate a useful comparison. We mentioned the need to provide mechanisms to assure consistency of the system during and after an adaptation, this has been further explored by Amano and Watanabe [29], at this stage, we do not aim at discussing consistency. Nevertheless by relying on a formal language we will be able to support consistency checks, hence, the selection of a best-fit formal language for dynamic adaptation can be considered an important contribution to the field of DA.

VIII. Conclusion

In this work we analysed three formal service-oriented languages with respect to their capabilities in modelling DA. Real time DA is an area of research that poses new challenges to software development, considering software that may adapt to changing conditions in the operational environment, where new services may be added as they become available, or cope with reconfiguration issues, all this at runtime and under time

constraints. DA has been proposed to provide solutions to these challenges. Proposing a methodology for the study of DA is still an open question. Formal methods have been in use for a long time in the computer science community and a number of new approaches and formal languages is available. Modelling DA with a formal language can provide precise answers to most of the existing questions and grant a better understanding of DA. This paper analyses three formal service-oriented languages. We first proposed some requirements on service-oriented languages for DA, second we presented definitions of the elements in DA, third we outlined a tollbooth scenario to exemplify some aspects of the languages, and finally we analysed the languages. The three languages have different strengths and weaknesses. JOLIE provides limited support for DA because there is no mechanism to add behaviour at execution time. PiDuce on the other hand, despite allowing for creation and substitution of channels, does not support modification or substitution of processes. PiDuce programs have a static structure and lack an adaptation manager. Both JOLIE and PiDuce lack a model for timeliness. COWS provides for the specification of service oriented applications and modelling of dynamic behaviour, furthermore it allows to model service substitution and adaptation by means of conditional choice and sequential composition (See Table IV). COWS is also intended to provide support for developing tools for checking that a given service composition follows desirable correctness properties and unexpected behaviours are avoided. These characteristics make COWS the best-fit option for modelling dynamic adaptation. A formal service-oriented approach to DA is still an open issue and this work represents a step forward in this direction.

ACKNOWLEDGMENT

This work was supported, in part, by Science Foundation Ireland grant $03/CE2/I303_1$ to Lero - The Irish Software Engineering Research Centre (*www.lero.ie*)

REFERENCES

[1] Business Process Execution Language for Web Services (BPEL4WS). http://xml.coverpages.org/bpel4ws.html.

[2] J. Han and A. Colman, "The four major challenges of engineering adaptive software architectures," *Computer Software and Applications Conference, 2007. COMPSAC 2007 - Vol. 2. 31st Annual International*, vol. 2, pp. 565–572, 24-27 July 2007.

[3] R. Hirschfeld and K. Kawamura, "Dynamic service adaptation," *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, pp. 290–297, March 2004.

[4] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[5] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "A taxonomy of compositional adaptation," Dept. Computer Science and Engineering, Michigan State University, Tech. Rep. MSU-CSE-04-17, 2004.

[6] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro, "SOCK : A calculus for service oriented computing," *Proceedings of Service-Oriented Computing Ű ICSOC 2006*, vol. 4294/2006, pp. 327–338, 2006.

[7] C. Guidi, "Formalizing languages for service oriented computing," Ph.D. dissertation, Department of computer Science, University of Bologna, 2007.

[8] C. Guidi and F. Montesi, "Reasoning about a service-oriented programming paradigm," *Electronic Proceeedings in Theoretical Computer Science*, vol. 2, p. 67, 2009.

[9] F. Montesi, C. Guidi, and G. Zavattaro, "Composing services with JOLIE," in *ECOWS '07: Proceedings of the Fifth European Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–22.

[10] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro, "On the Interplay Between Fault Handling and Request-Response Service Invocations," *8th International Conference on Application of Concurrency to System Design, 2008. ACSD 2008.*, pp. 190–199, 2008.

[11] C. Guidi and R. Lucchi, "Formalizing mobility in service oriented computing," *Journal of Software*, vol. 2, no. 1, pp. 1–13, 2007.

[12] F. Montesi, C. Guidi, I. Lanese, and G. Zavattaro, "Dynamic fault handling mechanisms for service-oriented applications," *IEEE Sixth European Conference on Web Services, 2008. ECOWS '08.*, pp. 225–234, Nov. 2008.

[13] A. L. B. Jr., C. Laneve, and L. G. Meredith, "Piduce: A process calculus with native xml datatypes," in *EPEW/WS-FM*, ser. Lecture Notes in Computer Science, M. Bravetti, L. Kloul, and G. Zavattaro, Eds., vol. 3670. Springer, 2005, pp. 18–34.

[14] S. Carpineti, C. Laneve, and L. Padovani, "PiDuce – a project for experimenting web services technologies," *Science of Computer Programming*, vol. 74, no. 10, pp. 777 – 811, 2009.

[15] A. Lapadula, R. Pugliese, and F. Tiezzi, "A Calculus for Orchestration of Web Services," in *Proc. of 16th European Symposium on Programming (ESOP'07)*, ser. Lecture Notes in Computer Science, vol. 4421. Springer, 2007, pp. 33–47.

[16] A. Lapadula, R. Pugliese, and F. Tiezzi, "Cows: A timed service-oriented calculus," in *Proc. of 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, ser. Lecture Notes in Computer Science, vol. 4711. Springer, 2007, pp. 275–290.

[17] "Calculus for Orchestration of Web Services (COWS)," http://rap.dsi.unifi.it/cows/.

[18] R. De Nicola, D. Gorla, and R. Pugliese, "On the expressive power of klaim-based calculi," *Theoretical Computer Science*, vol. 356, no. 3, pp. 387–421, 2006.

[19] R. De Nicola, D. Gorla, and R. Pugliese, "Basic observables for a calculus for global computing," *Information and Computation*, vol. 205, no. 10, pp. 1491–1525, 2007.

[20] M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro, "SCC: A service centered calculus," in *Web Services and Formal Methods*, ser. Lecture Notes in Computer Science, M. Bravetti, M. Núñez, and G. Zavattaro, Eds., vol. 4184. Springer, 2006, pp. 38–57.

[21] M. H. ter Beek, A. Buccharione, and S. Gnesi, "Formal methods for service composition," *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, 5, pp. 1–10, 2007.

[22] K. Geihs, "Selbst-adaptive software," *Informatik-Spektrum*, 2007, 0170-6012 (Print) 1432-122X (Online).

[23] M.-T. Segarra and F. André, "A framework for dynamic adaptation in wireless environments," in *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 336.

[24] A. Ferrara, "Web services: a process algebra approach," in *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. New York, NY, USA: ACM, 2004, pp. 242–251.

[25] F. Abouzaid, "A mapping from pi-calculus into BPEL," in *Proceeding of the 2006 conference on Leading the Web in Concurrent Engineering*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2006, pp. 235–242.

[26] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede, "Formal semantics and analysis of control flow in ws-bpel," *Sci. Comput. Program.*, vol. 67, no. 2-3, pp. 162–198, 2007.

[27] C. Laneve and G. Zavattaro, "Foundations of web transactions." Springer, 2005, pp. 282–298.

[28] N. Mori and H. Kita, "Genetic algorithms for adaptation to dynamic environments - a survey," in *26th Annual Conference of the IEEE Electronics Society IECON 2000*, I. P. I. E. Conference), Ed., 2000.

[29] N. Amano and T. Watanabe, "Towards constructing component-based software systems with safe dynamic adaptability," in *International Workshop on Principles of Software Evolution (IWPSE)*, 2001.