HAL
open science

A Rule-Based Language and Verification Framework of Dynamic Service Composition

Willy Kengne Kungne, Georges-Edouard Kouamou, Claude Tangha

To cite this version:

Willy Kengne Kungne, Georges-Edouard Kouamou, Claude Tangha. A Rule-Based Language and Verification Framework of Dynamic Service Composition. Future internet, 2020, 12 (2), pp.23. 10.3390/fi12020023 . hal-02457316

HAL Id: hal-02457316

https://hal.science/hal-02457316v1

Submitted on 21 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

future internet

MDPI

Article

A Rule-Based Language and Verification Framework of Dynamic Service Composition

Willy Kengne Kungne [1,*], Georges-Edouard Kouamou [2] and Claude Tangha [3]

[1] Department of Computer Sciences, Faculty of Sciences, University of Yaoundé I, P.O. Box 812 Yaoundé, Cameroon
[2] Department of Computer Sciences, National Advanced School of Engineering, University of Yaoundé I, P.O. Box 8390 Yaoundé, Cameroon; georges.kouamou@polytechnique.cm
[3] Faculty of Information Technologies and Communication, Protestant University of Central Africa, P.O. Box 4011 Yaoundé, Cameroon; c.tangha@gmail.com
* Correspondence: kengnekungnewilly@yahoo.fr; Tel.: +237-675-978-859

Received: 8 December 2019; Accepted: 23 January 2020; Published: 26 January 2020

Abstract: The emergence of BPML (Business Process Modeling Language) has favored the development of languages for the composition of services. Process-oriented approaches produce imperative languages, which are rigid to change at run-time because they focus on how the processes should be built. Despite the fact that semantics is introduced in languages to increase their flexibility, dynamism is limited to find services that have disappeared or become defective. They do not offer the possibility to adapt the composite service to execution. Although rules-based languages were introduced, they remain very much dependent on the BPML which is the underlying technology. This article proposes the specification of a rule-based declarative language for the composition of services. It consists of the syntactic categories which make up the concepts of the language and a formal description of the operational semantics that highlights the dynamism, the flexibility and the adaptability of the language thus defined. This paper also presents a verification framework made of a formal aspect and a toolset. The verification framework translates service specifications into Promela for model checking. Then, a validation framework is proposed that translates the verified specifications to the operational system. Finally, a case study is presented.

Keywords: rule-based approach; service choreography; flexibility by change; adaptability; DSL; model checking

1. Introduction

Services are increasingly developed and published on the Web. Often, these services taken alone serve lightweight functionality. In many cases, a single service is not sufficient to satisfy user requests. So several services need to be combined to achieve a specific goal [1,2]. Several languages were proposed to define composite services. These languages rely for the most part on process models with the BPML language as the reference [2]. The latter provides structured and rigid blocks based on graphs which are then translated into static composite services.

The ability to manage the flexibility and the evolution of a composite service at runtime is an essential requirement. For example, a user in order to prepare its travel, needs to buy a flight ticket and reserve a room. The user get in touch with a travel agent which will communicate with a hotel service and an airline service. May be the travel agent make available the possibility to rent a car. This new service can be provided by a car rent service or one of the initial agent (hotel or airline). The new service will be added in the travel agent space without stop the running service. In addition, the car rent service is optional for the users who need it.

The changes in travel agent requirements have an immediate impact on the specification of the composite services. This situation can occur even at runtime since the services which participate to a composite can be removed or added any time.

Therefore, the need to have the flexible and adaptive services composition approach is a necessity. Several studies have been carried out to make them more adaptable during execution [1–6]. To this end, it is appropriate to offer languages that promote more flexibility (as soon as the services are specified) and adaptability (in the execution framework). To meet these requirements, rule-based declarative approaches were proposed because they promote reusability, adaptability and flexibility for the composition of services [7].

How can we take into account the user needs or the new services even during the execution of a composite service? The research question in this paper deals with the proposition of a language for the definition of flexible and adaptable services. Rather than providing runtime techniques to adapt to the changes [3], the required need consists of anticipating the changes by relying on a language which is purely declarative and rules oriented which facilitates a greater flexibility and adaptation by describing the execution schema of the composition (the *what*) rather than the execution path of the composition (the *how*). The proposed language here is based on the production rules with the purpose to operationalize the Guarded Attributed Grammar (GAG) model [8] through the specification of a service composition language. Basically, GAG models are designed principally in order to contribute to the modelling of artifact-driven workflows. They are declarative rules-based, data-oriented, user-centric, and distributed. The transposition of all these properties in the domain of the services composition makes it possible to have a more flexible composition environment because the language proposed in this article is purely declarative and based on the rules.

A composite service is defined by using grammar productions as composition rules. The left hand side of a production rule defines a new service that is obtained by assembling the entities on the right hand side. Any change in the description of the composition scheme is automatically taken into account when the affected rule is applied. The services are connected at runtime through the dependencies between their attributes. The composition is controlled by the data and the associated conditions so-called guards. The operational semantics which consists of a description of the services selection, the creation of the instances of the services and their refinements is presented using process algebra (pi-calculus) [9]. The choice of pi-calculus is supported by the fact of the distribution of peers in a network and they interactions through dynamic ports created at runtime.

The rest of the paper is structured as follows. Section 2 presents the state of the art on rule-based service composition languages. In Section 3, a declarative language for the composition of services is proposed, which consists of the syntax definition, the specification of the operational semantics and its verification. In Section 4, a verification framework of the proposed language is presented based on the PROMELA language and the SPIN tool. Section 5 describes a validation framework that presents a prototype for evaluation of our approach, this prototype consists of an editor (DSL) for the proposed language, model transformation rules to Promela/SPIN and an execution engine. Section 6 highlights the strengths of our work compared to the existing in the field of the composition of services. Section 7 concludes the work and presents some future directions.

2. A State of the Art on Ruled Based Specification of Service Composition

Traditional languages proposed for specifying web services composition are process-based hence their imperative character on the one hand. On the other hand, they are very related to the underlying technology.

Languages that are standards such as BPEL [10], BPML [11], WS-CDL [12] or WSCI [13] propose to define composite services in the form of structured blocks thus making their flexibility and adaptation difficult. Several other languages and tools have been proposed based on these standardized languages to improve their flexibility [1,2,14]. The other disadvantage of these languages is that the processes are centralized and difficult to modify during execution. Moreover, the composite services are described

by WSDL [15] interfaces therefore the orchestration of other types of services that are not compliant with WSDL is not possible. The level of abstraction of the specification of the composition is quite low.

To overcome these difficulties, some languages propose increasing the autonomy of the peers and effective distribution [16,17]. They also present the semantics of the composition by offering the possibility of describing and reasoning about the services during the execution [18,19]. These languages based on semantics are excellent to discover, select and automatically compose services.

However, their flexibility is limited to search the missing services or to build a composition plan based on the query of user and a predefined planning system. It is difficult to add new requirements to a specification of composite service when it is executed.

Instead, the rules paradigm has been presented as a declarative approach. It presents the following advantages [7,20,21]:

- Flexibility: compositions based on rules are more flexible than BPML-type compositions. It is able to pursue other execution paths in case where a particular execution path fails without to redefine the composite and redeploy it;
- Adaptability: given the declarative nature of the rule-based service compositions, they can be update to adapt to specific situations, for example in terms of external services or platform deployment;
- Reusability: Since rules are isolated from the context of the business process, they can be reused more easily in other service application contexts;
- Formal semantics: languages based on rules exploit a logical and/or mathematics set. Formal approaches allowing reasoning have been proposed [22–24] but all use the WS-BPEL process type for their implementation.

Considerable efforts have been invested in providing the rule engines to support flexible and adaptable service compositions.

In [4,25], a service-oriented rule engine was introduced, allowing access to business rules by invoking the RuleML engines. [23] introduces an alternative service execution environment in which rules can be defined and then injected into WSDL specifications, after which they can be deployed to a service executor.

Reference [26] defines in the form of clauses *if..then*, the structure, the data and the constraint rules under the basis of elements such as Message, Event, Condition, Provider, Flow, Role, and Activity. This is a first step for the flexible composite service specification but it is presented as an extension of the BPEL notations. To separate the business rules from the BPEL code, Charfi et al. suggested an aspect-based style (AO4BPEL) [27].

Reference [7] argues that business rules can be used in a service composition without the need for a BPEL framework. This increases the adaptability of the orchestration. At the deployment level, a *CA* (Condition-Action) rule engine supports rules-based service composition. To obtain the composite, an analysis of the registry (containing WSDLs) of the services is performed in order to extract the dependencies between services and to build the CA rules. The business rules and the constraints added to the CA rules. Although using the rules to build the composites, this approach has an abstraction level of the rules which is quite low (linked to WSDL). Moreover, it focuses, as do the previous ones, on the orchestration thus leaving the distribution and interaction in the background.

This study adopts an independent approach of the block structuration such as BPML. We promote describing a composition service completely with rules, using the GAG formalism. Reference [8] to intentionally specify the composition of the services. The intentional definition of services composition makes them abstract, which increases the flexibility because the link between the intentions and the implementation is done when a rule is enacted. Any update is possible on a rule and it is taken into account the next time the rule is enacted. The advantage of this property allows conforming the rules with the user needs even at runtime by adapting the rules to the new requirements. Moreover, the use of rules in a service composition language brings the formal intuitive semantics, flexibility, adaptability, and reusability.

3. Declarative Formalism for the Service Composition

One of the main challenges of software engineering is dealing with the changes (social changes, user requirements changes, etc.). Concerning the aspect of service composition, one talks about flexibility by changing, which means modifying the structure of a composite service even at runtime [28]. We propose, in this section, a service specification formalism that uses productions of a context-free grammar [29]. This so-called intentional specification, because it remains abstract, is constituted at the right hand side (RHS) of the description of the services that concur to solve the left-hand side (LHS) service. In our approach, the composition scheme remains as abstract as possible, because even during execution, only the concerned rule is enacted and instantiated.

3.1. Basic Definition

A service is defined within the frame of an activity. An activity is a tuple

$$\mathrm{A} = (id, S, C)$$

where

- **Id** is a unique identifier of the activity (for example, its name);
- **S** is the execution schema that describes the services $\{t_0, t_1 ... t_n\}$ to be performed to resolve the activity;
- **C** is the context indicating information from the environment in which the activity is performed. This information can contribute to the modification of the execution scheme of an activity. $C = (I, O)$ where I is the inputs getting from the environment and O the outputs returned to the environment at the end of the execution of the activity.

Each service s ∈ S can be defined in one of the following forms:

$$\begin{aligned} s(\mathrm{c_0...c_n})\langle \mathrm{r_0...r_m}\rangle \rightarrow\ & \mathrm{s_1}(\mathrm{i_{1i}...i_{1u}})\langle \mathrm{o_{1i}...o_{1k}}\rangle \ ... \\ & \mathrm{s_j}(\mathrm{i_{j1}...i_{jo}})\langle \mathrm{o_{j1}...o_{jh}}\rangle \end{aligned} \quad (1)$$

$$s(\mathrm{c_0...c_n})\langle \mathrm{r_0...r_m}\rangle \rightarrow \quad (2)$$

In this respect, we distinguish composite services if the right hand side of the rule is not empty, and elementary services otherwise. A composite service depends on the services which appear on the right hand side for its completion in other words the RHS services aggregate the service that appear at the LHS. An elementary service provides access to Internet-based application via a well-known protocol such as SOAP or REST architecture. This concept of service will help to ensure interoperability among the peers. For each type of rule, a function of correspondences that defines the semantic rules between the parameters can be associated. Let A be an activity accommodated by a peer h. If A is defined as:

$$\begin{aligned} s(\mathrm{c_0...c_n})\langle \mathrm{r_0...r_m}\rangle \rightarrow\ & \mathrm{s_1}(\mathrm{i_{1i}...i_{1u}})\langle \mathrm{o_{1i}...o_{1k}}\rangle \\ & ...\ \mathrm{s_j}(\mathrm{i_{j1}...i_{jo}})\langle \mathrm{o_{j1}...o_{jh}}\rangle \end{aligned}$$

where s_1, .., s_j make up the service s, the c_k are the input parameters to s and the r_k are the output parameters, the i_{kj} are the input parameters of the RHS and the o_{kj} are the output parameters.

The semantic rules are given by the dependencies between attributes which make possible to apply the substitutions

$$\begin{cases} i_{kl} = f(i_{ij}, o_{mn}, c_j) with\ k \neq\ i,\ l\ \neq\ j \\ r_k = f(o_{mn}) \end{cases}$$

Where the inputs of the RHS depends on the inputs of the RHS of other services, the outputs of the RHS and the inputs of LHS of s. The outputs of the LHS are functions of the outputs of the RHS. f materializes this dependency Example:

$$travelService(username, date, destination)\langle numVol, hotelAdress\rangle \rightarrow$$
$$. \quad flightBooking(username, date, destination)\langle numVol\rangle$$
$$. \quad hotelReservation(username, date)\ \langle hotelAdress\rangle$$

$numVol$ of $travelService$ corresponds to $numVol$ to $flightBooking$. Other examples can be taken for this $travelService$ service.

The semantic rules are also expressed by the conditions on the input and the output parameters which serve as guards and post-conditions.

This so-called intentional definition is materialized during its instantiation. The service instance takes place according to the availability of the data (parameters). When the composition scheme of an activity is updated, the modification will be applied next time the associated rule be enacted. The semantic rules may involve several variables which belong to different peers. At this moment a private dynamic channel is opened for the interaction between the peers. These are the reasons why the pi-calculus is chosen in the next section to specify the composition language. In fact, one can notice that calculi based on the pi calculus provide primitives for describing and analyzing global distributed infrastructure, focusing on process migration between peers, process interactions via dynamic channels, and private channel communication [9]. In the process algebra (pi-calculus) [9], the processes execute actions that can have three forms: the sending of a message over x channel (written $\overline{x}$), the receiving of a message over x channel (written x), and silent actions (written τ), whose details are unobservable. Send and receive actions are called actions of synchronization, since communication is highlighted when the processes synchronize. The transition notion represents the executing a process expression. Intuitively, the transition relation indicates how to perform one-step of the process execution. Note that since there may be many ways to execute a process, the transition is basically non-deterministic. The transition of a process A into a process B by performing an action α is indicated A $\xrightarrow{\alpha}$ B. The action α is the observation of the transition.

The description of the services thus defined and their behaviors are based on primitives of instantiation, refinement and exchange of messages which can also be found in more abstract languages such as process algebras. This permits us to describe the structure of a service and present formally how a service is defined and how it interacts in its environment. In the following, we describe formally the syntax and the semantics of a service composition language.

3.2. *A Formal Syntax and Semantic of the Service Composition Language: GSLang*

The language is called GSLang and was introduced in [30]. This section presents GSLang and shows how the defined services can be resolved. The elements of the GSLang language are:

3.2.1. Variables and Terms

A **variable** is a letter; it can contain a value. The **terms** are values, variables, defined variables (assignment), functions on the terms or Boolean expressions.

In this paper we define, $\overline{\overline{x}}$ for tuple (x_1,... x_n).

$$\begin{array}{rcl} t & ::= & x(variable) \\ & | & u(value) \\ & | & x_r(defined\ variable) \\ & | & f(t_0 ... t_n) \end{array}$$

To the basic syntax of pi-calculus, we add Boolean expressions to check the activation and validation of a service. **Boolean expressions** when specified and evaluated gives a Boolean value. In other words it is a logical expression on the variables. They serve as guards when activating a service. In this case, they are pre-conditions; they control the triggering of a service. At the end of the execution of a service, there may be conditions to be satisfied. These are post-condition.

$$\begin{aligned} \mathrm{e_b} \quad ::= \quad & true \mid false \mid \mathrm{t_j} \leq \mathrm{t_i} \\ & \mid \mathrm{t_i} = \mathrm{t_j} \mid \mathrm{e_b} \mid \mathrm{e_b} \wedge \mathrm{e_b} \mid \mathrm{e_b} \vee \mathrm{e_b} \end{aligned}$$

The **assignment** affects a value to a variable.

$$\mathrm{x_r} \quad ::= \quad \epsilon \mid \mathrm{x_r}\ (x \leftarrow t)(value\ assignment)$$

A parameter is an output or an input variable related to a service. The scope of a variable is related to the associated service. The context of activity is characterized by input variables and output variables. These variables are free and unique throughout the system. One notes that a variable is defined or resolved when a value is assigned to it (x ← u). A variable is define once and can be used several times.

3.2.2. Service and Service Instance

A **service** is defined by a unique identifier, output variables (output parameters), input variables (input parameters), post-conditions, guards and a location. A service can depend on other services.

$$\begin{aligned} S \ ::= \ & id(\ \overline{\overline{x}}, \overline{\mathrm{e_b}}^{x})\ \langle \overline{\overline{y}}, \overline{\mathrm{e_b}}^{y} \rangle\ [\alpha] \mid \\ & id(\ \overline{\overline{x}}, \overline{\mathrm{e_b}}^{x})\ \langle \overline{\overline{y}}, \overline{\mathrm{e_b}}^{y} \rangle\ [\alpha] \rightarrow \mathrm{S_1 ... S_n} \end{aligned}$$

A service is elementary or composite. It is characterized by output parameters $\overline{\overline{y}}$, input parameters $\overline{\overline{x}}$, possibly the effects on the output parameters $\overline{\mathrm{e_b}}^{y}$, possibly preconditions on input parameters $\overline{\mathrm{e_b}}^{x}$ and an identifier (its name). It can be composed of other services $\mathrm{S_1 ... S_n}$. α represents the location of service. It should be noted that α may be unnecessary for services on the RHS if they are implemented in the same user space as the services from the LHS.

For reasons of readability, we prefer the previous notation. In pi-calculus notation, it corresponds to:

$$\begin{aligned} S \ ::= \ & [\overline{e_b}^{x}]\ \alpha\ (id, \overline{\overline{x}}, p)\ .\ [\overline{e_b}^{y}]\ p!\overline{\overline{y}}\ \mid \\ & [\overline{e_b}^{x}]\ \alpha\ (id, \overline{\overline{x}}, p)\ .\mathbf{(}vp_1\ \alpha\ \langle id_1, \overline{\overline{x}}_1, p_1 \rangle\ \mid \\ & ...\ \mid\ S_i ...\ \mid\ vp_n\ \alpha\ \langle id_n, \overline{\overline{x}}_n, p_n \rangle\mathbf{)}\ .\ [\overline{e_b}^{y}]\ p!\overline{\overline{y}} \end{aligned}$$

The S service expects $\overline{\overline{x}}$ as the parameter, executes and returns $\overline{\overline{y}}$. It receives the data on the public port of the peer where it is hosted. when the RHS is present, it calls the services it contains to build y. The services on the RHS can be executed in parallel if the data are independent of each other or in sequence if there is dependency hence the parallel operator(|) inside the brackets in bold. Once a call of service performed, we get the service instances. These notations are also used to describe the instances.

The **Service instance** or **Artifact** is the instantiation of a service. This makes it possible to follow the execution of the service.

$$
\begin{array}{rcl}
I & ::= & id(\ \overline{\overline{x}}, \overline{e_b}^{x})\ \langle \overline{\overline{y}}, \overline{e_b}^{y} \rangle\ [\alpha]\ | \\
 & & id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}, \overline{e_b}^{y} \rangle\ [\alpha]\ | \\
 & & id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}_r, \overline{e_b}^{y} \rangle\ [\alpha]\ | \\
 & & id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}, \overline{e_b}^{y} \rangle\ [\alpha] \rightarrow \mathrm{I}_1 ... \mathrm{I}_n\ | \\
 & & id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}_r, \overline{e_b}^{y} \rangle\ [\alpha] \rightarrow \mathrm{I}_1 ... \mathrm{I}_n
\end{array}
$$

A service instance has several configurations:

- the input and the output parameters are not yet resolved;
- resolved input parameters and output parameters not yet resolved;
- resolved input and output parameters.

Each element that appears on the right, when it exists, can have one of the three previous configurations. The parameters of the service instances are resolved as they are executed.

3.2.3. Action

An **action** is a send message, a receive message or a silent interpretation of the service instances.

$$
\begin{array}{rcl}
act & ::= & vp\ \overline{\alpha}\ \langle M, p \rangle \\
 & | & \alpha(M, p) \\
 & | & p(M) \\
 & | & \overline{p}\ \langle M \rangle \\
 & | & r
\end{array}
$$

The **actions** define the communication to be performed between the services execution spaces:

- $vp\ \overline{\alpha}\ \langle M, p \rangle$ allows, firstly, the creation of a variable representing the private port (p), then send a message M and p to the public port α of a remote space;
- $\alpha(M, p)$ receiving a message M and a variable p on the public port α;
- $p(M)$ receive a message M on a private port p;
- $\overline{p}\ \langle M \rangle$ sending a message M on a private port p;
- r silent action which consists of interpreting the services parameters using semantic rules. No communication with the environment.

3.2.4. Message

Messages are variables that are exchanged on the network. It contains global variables (context variables). It is composed of defined variables and/or not defined variables. There are two types of messages: request message (variables in defined inputs and variables in undefined output) and response message (defined input variable and output variable defined).

$$M ::= \ \overline{\overline{x}}_r\ \overline{\overline{y}}\ id\ \ (request)\ |\ \overline{\overline{x}}_r\ \overline{\overline{y}}_r\ \ (response)$$

A **sending message** (request) comprises 3 parts: resolved input $\overline{\overline{\mathrm{x}}}_\mathrm{r}$, outputs to be resolved $\overline{\overline{\mathrm{y}}}$ and the identifier of the service to which the request is intended. A **response message** consists of 2 parts: resolved inputs $\overline{\overline{\mathrm{x}}}_\mathrm{r}$ and resolved outputs $\overline{\overline{\mathrm{y}}}_\mathrm{r}$. As we will see in the next section, the response is transmitted along a private port created during the request, hence the absence of the service identifier.

3.3. Behavioral Description

The following operational semantics describes the mechanisms for resolving services, which is divided into several fundamental operations: sending and receiving of the message, refinement, instantiation and choice of services.

The execution space or **peer**(Σ) contains services, instances of services and is characterized by its location. Let us denote by I_s the set of service instances, S_s the set of services and α its address (main port or location). Thus the space of execution is characterized by Σ = (S_s, I_s, α). In the following, the different operations to be applied on an execution space for the resolution of the services, $fn(e)$ is set of bound variables of the entity e.

Intanciation

$$\frac{\Sigma' = \Sigma \; U \; I, \; p \notin fn(\Sigma)}{\Sigma : S = id(\overline{\overline{x}})\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{\alpha(M,p)} \Sigma' : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]} C_1$$

$$\frac{\Sigma' = \Sigma \; U \; I, \; p \notin fn(\Sigma)}{\begin{array}{c}\Sigma : S = id(\overline{\overline{x}})\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{\alpha(M,p)} \Sigma' : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \\ \rightarrow \mathrm{S_1...S_n} \qquad\qquad \rightarrow \mathrm{I_1...I_n}\end{array}} C_2$$

When Σ receives on its main port α the message named M and the variable named p, it finds the corresponding service named S and creates the instance named I with the input which are defined $\overline{\overline{x}}_r$ and the outputs $\overline{\overline{y}}$ awaited. I is added to Σ which becomes Σ′. If no service is found, the operation is not applied.

C2 is the extended version of C1, the service found is composed.

Response

$$\frac{}{\Sigma : I \xrightarrow{p\,\langle M\rangle} \Sigma : I} Resp$$

Response on p (private port) of a previously sent request. M=$\overline{\overline{x}}_r \;\; \overline{\overline{y}}_r$

Request

$$\frac{\begin{array}{l} I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \; or \\ \;\; = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \mathrm{I_1...I_n} \; or \\ \;\; = I_0 \rightarrow I_1...id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]...I_n \end{array}}{\Sigma : I \xrightarrow{vp\;\overline{\alpha}\;\langle M,p\rangle} \Sigma : I} Req$$

The I of Σ sends the M on α. This does not change the state of the execution space; M is constructed from parameters of the instance to be concretized.

Refinement

$$\frac{}{\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{r} \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]} R1$$

$$\frac{}{\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{p(M)} \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]} R2$$

$$\frac{\overline{\overline{x}}' \subseteq \overline{\overline{x}}}{\begin{array}{l}\Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \\ \quad I_1...id_i(\overline{\overline{x}}')\langle\overline{\overline{y}}'\rangle[\alpha'] \\ \quad ...I_n\end{array} \xrightarrow{r} \begin{array}{l}\Sigma : I = id_0(\overline{x}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \\ \quad I_1...id_i(\overline{\overline{x}}_r')\langle\overline{\overline{y}}'\rangle[\alpha'] \\ \quad ...I_n\end{array}} R3$$

$$\frac{\overline{\overline{x}}'' \subseteq \overline{\overline{y}}'}{\begin{array}{ll} \Sigma : I = I_0 \rightarrow I_1 ... & \Sigma : I = I_0 \rightarrow I_1 ... \\ id_i(\overline{\overline{x'}}_r)\langle\overline{\overline{y'}}_r\rangle[\alpha'] ... & id_i(\overline{\overline{x'}}_r)\langle\overline{\overline{y'}}_r\rangle[\alpha'] ... \\ id_j(\overline{\overline{x''}})\langle\overline{\overline{y''}}\rangle[\alpha''] ... I_n & id_j(\overline{\overline{x''}}_r)\langle\overline{\overline{y''}}\rangle[\alpha''] ... I_n \end{array}} R4 \quad \xrightarrow{r}$$

$$\frac{}{\begin{array}{ll} \Sigma : I = I_0 \rightarrow I_1 ... & \Sigma : I = I_0 \rightarrow I_1 ... \\ id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] ... I_n & id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha] ... I_n \end{array}} R5 \quad \xrightarrow{p(M)}$$

$$\frac{\overline{\overline{y}} \subseteq \cup \overline{\overline{y}}_i}{\begin{array}{ll} \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow & \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha] \rightarrow \\ id_1(\overline{\overline{x}}_{1r})\langle\overline{\overline{y}}_{1r}\rangle[\alpha] & id_1(\overline{\overline{x}}_{1r})\langle\overline{\overline{y}}_{1r}\rangle[\alpha] \\ ...id_n(\overline{\overline{x}}_{nr})\langle\overline{\overline{y}}_{nr}\rangle[\alpha] & ...id_n(\overline{\overline{x}}_{nr})\langle\overline{\overline{y}}_{nr}\rangle[\alpha] \end{array}} R6 \quad \xrightarrow{r}$$

The refinement of an instance is the materialization of not yet defined parts. The action is silent or the receipt of a response on a private port.

R1: Calling a primitive service (automatic or manual)

R2: receive information on a private port for an instance of service.

R3, R4 and **R6:** Allow the definition of the parameters of some service instances to the RHS from the parameters already defined.

R5: Upon receipt of a response, materialize the part of the service instance that made the request.

Local choice of Service (LoCh)

$$\frac{id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \;\; match \;\; to \;\; I'_0 \rightarrow I'_1 \;\; ... \;\; I'_n}{\begin{array}{ll} \Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \rightarrow & \Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \rightarrow \\ I_1 ... id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle ... I_n & I_1 ... [I'_0 \rightarrow I'_1 \;\; ... \;\; I'_n] ... I_n \end{array}} \xrightarrow{r}$$

select a service locally.

The focus is on the data that influences the choice of services, their creations and their refinements. The execution flow depends on the availability of the values of the input and output variables. In a schema of service composite, two tasks execute in parallel if the inputs of one do not depend on the outputs of the other and in sequence if the entries of one depend on the outputs of the other.

3.4. Correspondence with Traditional Process Creation Activities

The *GSLang* is based on the rules that describe a service by specifying its name, input and output parameters, semantic rules (describing the correspondence between attributes) and services to solve it. According to the dependency of the attributes parameters and the enchainment of services within the rules, we have sequential instructions, conditional expressions, loops, and exceptional case handling.

3.4.1. Sequence

It can happen only in a rule defining a service s, the input parameters to a service s_2 correspond to the output parameters of a service s_1. Then services s_1 will execute before service s_2. For e.g., $s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s_2(y')\langle y\rangle$

3.4.2. Condition

In a Σ execution space, a service may have several different definitions, so the conditional expression will guide the choice of one over the others. For example if $a > 0$ then the first definition will be chosen otherwise, if $a = 0$ it is the second one.

$$[a > 0]s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s_2(y')\langle y\rangle$$

$$[a = 0]s(x)\langle y\rangle \rightarrow$$

3.4.3. Loop

In the sequence of a rule defining a service s, we can have the service s itself in its RHS or in the RHS of a service present in the RHS of s. Defined thus, we have a loop. For example

$$[x > 0]s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s(y')\langle y\rangle$$

$$[x = 0]s(x)\langle y\rangle \rightarrow$$

It is very difficult in practice to determine that the service will run until the end. To do this, we transform the service specifications into Promela/SPIN in order to validate them.

3.4.4. Parallele (Fork/Join)

If the two services s_1 and s_2 present on the right side of a service have nothing in common, then s_1 and s_2 can run at the same time. It can happen that the output of s depends on the outputs of s_1 and s_2 as in the example. $s(x)\langle y_1, y_2\rangle \rightarrow s_1(x)\langle y_1\rangle s_2(x)\langle y_2\rangle$

3.4.5. Exceptional Case Handling

As the specification of services is declarative, adding specific rules with conditions can help manage exceptional cases. This declarative description allows you to add rules at any time to handle exceptional cases.

3.5. Service Resolution

The operational semantics defined above consist in creating and materialize the service instances dynamically at runtime according to the input and output parameters and the port created during execution. In the system a service instance I is resolved if all these parameters are defined (input and output). Any action on a resolved service does not change it. i.e.,

I is resolved $\iff I^*$

The semantics of this operator are as follows: if I^* and I $\xrightarrow{\alpha} I'$ then I'^*.

Theorem 1. *A service instance I to which are attached the input variables $\overline{\overline{x}}$ and the output variables $\overline{\overline{y}}$ is resolved if all the associated service instances I_i are resolved i.e.,*

$$\frac{I \rightarrow I_1 ... I_n,\ I_1^*,\ I_2^* ...,\ I_n^*}{I^*} where\ I_i^* = id(\overline{\overline{x}}_{ir})\langle\overline{\overline{y}}_{ir}\rangle$$

Proof. If the instance is simple i.e., it is defined by $I \rightarrow$, then I is resolved if the values are assigned to the input parameters x_r and the output parameters y_r.

If the instance is composite i.e., $I \rightarrow I_1 ... I_n$

- A instance like $I \rightarrow I_1$, by definition, $x \subseteq x_1$ and $y \subseteq y_1$ if I_1 is resolved i.e. we have x_{r1} and y_{r1} then x and y are defined i.e. x_r and y_r.

- A instance like $I \rightarrow I_1 ... I_n$ by definition, The input (resp. output) variables of the LHS x (resp. y) depends on the input (resp. the output) parameters of the RHS: $x \subseteq \cup_i x_i$ (resp. $y \subseteq \cup_i y_i$). if $\forall_i I_i$ is resolved i.e. we have the x_{ir} and y_{ir}, $\Rightarrow$ we have x_r and y_r.

□

Proposition 1. *The operational semantics generate a $\perp$-transitions system.*

Proof. Let a labeled $\perp$-transition system defined by a n-tuple $(\Sigma, S, \rightarrow, \perp)$ where

- Σ is a set of **actions** which label the transitions;
- $Q = S \cup I$ is the set of states where S is the set of **services** and I is the set of **instances**;
- $\delta \subseteq Q * \Sigma * Q$ the transition function
- $\perp \in S * bool$ is a termination predicate such that $\perp (s) = true$ and $s \xrightarrow{\alpha} s\prime$ then $\perp (s\prime) = true$

□

The transition system describes the behavior of a service when it is chosen in a user space. In Figure 1, all the operations of the operational semantics are used to describe how an instantiated service will progress to be solved.

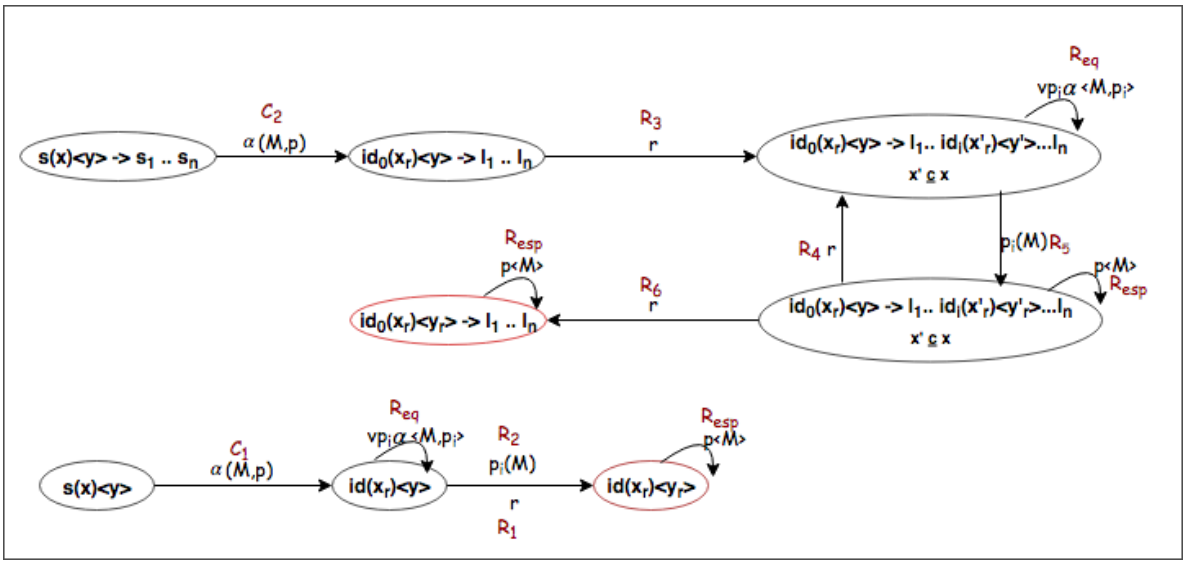

Figure 1. Progression of an instantiated service. Each transition is a rule of the operational semantics presented in Section 3.3.

The operations convert the system from one state to another through attribute interpretation (silent action), sent and received messages. These latter correspond in the transition system to labelled edges. The services and the instances are the nodes.

When we look at the instantiation rule C1, if the system containing a service of the form id($\overline{\overline{x}}$)$\langle\overline{\overline{y}}\rangle$ (node of the transition system) receives a message ($\alpha(M, p)$) (edge in the transition system) then the system creates an instance of the form $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle$. When the instance $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle$ is created, one can call the related service using the request $vp_i\alpha\langle M, p_i\rangle$ (rule Req). The receiving of the response to this $p_i(M)$ request (rule R2) allows refining the initial service thus defining the output $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle$.

On the other hand, for a composite service, applying rule C2 when receiving a message on the public port of a peer $\alpha(M, p)$, creates the instance with the values of the services of the LHS defined $(id_0(\overline{\overline{x}}_r)\langle y\rangle \rightarrow I_1..I_n)$. Therefore a refinement can be applied (rule R3) which allows through the semantic rules to define certain parameters of the RHS. At this point, the RHS services can be applied, requests $vp_i\alpha(M, p_i)$ are sent to another peer, responses received on private ports $p_i(M)$ (R5 rule) sent

during requests. Refinements are applied (rules R4 or R6). When all the outputs are set then we will reach the final state where the outputs of the LHS service are defined.

The verification of this semantics is demonstrated using SPIN tools [31]. The pi-calculus specification is translated into Promela and SPIN is used to validate the proposed model. We want to show that if the system is well specified, each instantiated service in the system will run until the end.

4. Verification Framework

The process of verifying and validating the above mentioned specification is described in Figure 2. It consists of the two phases:

- (i) Translate a service definition to a Promela specification;
- (ii) Simulate and verify the Promela specification to validate the properties.

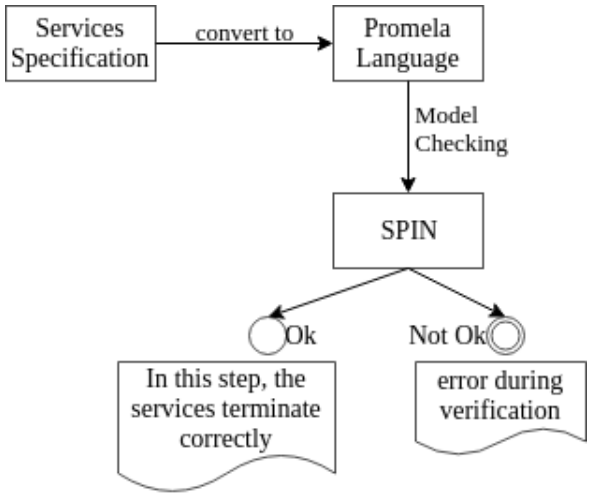

Figure 2. Verification process: the service specification is converted into PROMELA code for model checking.

4.1. *Translate Service Specification to Promela Language (Translation Template)*

The Promela language is an abstract specification language of a system. It models the synchronization and coordination of processes. Our choice was based on Promela, because it allows in a message to send a channel a bit like in the pi-calculus [32,33]. A message contains a dataset as well as a port on which the remote process will transmit its response.

Table 1 presents the correspondences between pi-calculus concepts and Promela code.

Table 1. Transformation rules from theoretical aspect to Promela language. Variables are translated into mtype, ports into channels, services into block structures and roles into processes.

Service Specification	Promela Code	Comment
variable, value and port	mtype, byte, chan	use *mtype* or *byte* to define variable and value And *chan* to define port
$x \leftarrow u \equiv x_r$	x = u	assignment the value (u) to the variable (x)
$\vec{e}_b^x$	$==, <, >, <=, !=, >=$	promela conditional expressions on $\overline{\overline{x}}$ for e.g., $x_1 == x_3$, $x1 > u$, $x_1 != x_3$. We will note them by **c**.
$s(\overline{\overline{x}}, \vec{e}_b^x)\ \langle \overline{\overline{y}}, \vec{e}_b^y \rangle [\alpha]$	**begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** $::(s == id\ \ \&\&\ \ c$ $\&\&\ \ \overline{\overline{x}})$ $\rightarrow p!\overline{\overline{y}};$ **goto begin;** **fi**	where α is the public port of user space(peer) containing **s** in the service specification, **p** is the private port contained in the request and used like the return channel. The Promela code translates the receipt of $s, \overline{\overline{x}}$ *and* p on α, the choice of the appropriate implementation according to the guard and the input parameters. it returns $\overline{\overline{y}}$ on **p**.
$s(\overline{\overline{x}}, \vec{e}_b^x)\ \langle \overline{\overline{y}}, \vec{e}_b^y \rangle [\alpha] \rightarrow$ $s_1(\overline{\overline{x}}_1)\ \langle \overline{\overline{y}}_1 \rangle [\alpha_1]$ $s_2(\overline{\overline{x}}_2)\ \langle \overline{\overline{y}}_2 \rangle [\alpha_2]$ $s_3(\overline{\overline{x}}_3)\ \langle \overline{\overline{y}}_3 \rangle$ ... $s_n(\overline{\overline{x}}_n)\ \langle \overline{\overline{y}}_n \rangle [\alpha_n]$	**begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** $::(s == id\ \ \&\&\ \ c$ $\&\&\ \ \overline{\overline{x}}) \rightarrow$ $\alpha_1!s_1, \overline{\overline{x}}_1, p_1$ $p_1?\overline{\overline{y}}_1;$ $\alpha_2!s_2, \overline{\overline{x}}_2, p_2$ $p_2?\overline{\overline{y}}_2;$ y_3 //define y_3 ... $\alpha_n!s_n, \overline{\overline{x}}_n, p_n$ $p_n?\overline{\overline{y}}_n;$ $p!\overline{\overline{y}};$ **goto begin;** **fi**	Meaning that upon receipt of $s, \overline{\overline{x}}$ *and* p on α, the choice of the appropriate service is made according to the input parameters and the guard (c). Then the services on the right side are processed in parallel or sequentially depending on whether the outputs correspond to inputs or not. Promela process is a peer. Private ports (p1, p2 ... pn) are created within the process representing the peer. Finally, sends the final response $\overline{\overline{y}}$ on channel **p**.
$\Sigma = (S_s, I_s, \alpha)$	chan α = [k] of {mtype,mtype chan}; **Proctype peerName(){** chan p1 = [2] of {mtype,mtype,chan}; ... chan pn = [2] of {mtype,mtype,chan}; **begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** $::(c_1) \rightarrow block_1$ **goto begin;** $::(c_2) \rightarrow block_2$ **goto begin;** ... $::(c_k) \rightarrow block_k$ **goto begin;** **fi** }	The promela process is a peer. The public ports are globally defined channels, private ports are local. The different services are the alternatives of the if instruction. We do not differentiate services from their instances. The code promela will be simply executed

4.2. Model Checking

The processes generated in Promela are simulated and verified using SPIN. When a new service is added, it is transformed into Promela and verified. The client process sends a request and waits for a result. When the client process receives the result, then the termination property for the called process (service) is verified. When for a verified service, all private ports receive the data, then the service terminates. The property of termination is therefore verified. In LTL (Linear Temporal Logic) this means: $[\,]\,(p?x)$ meaning that when a private port p is created, it always receives data. p being a created private port when running a service s and x being the received data.

The formal verification of the model contributes to the demonstration of the soundness property which is too difficult to proof theoretically [8].

Table 2 describes an example of services and their transformations into Promela distributed on three user spaces. The Promela transformation is then simulated. Three scenario are presented

scenario 1. The services were correctly specified (Figure 3); The client process calls the service s from *space1* on his public port. The *space1* process receives and instantiates s. s calls s_1 in *space2* by creating private port 4 on the $figure$, s_2 in *space3* by creating private port 5 on the figure and define y_3 locally. Finally, the client process receives the reponse awaited.

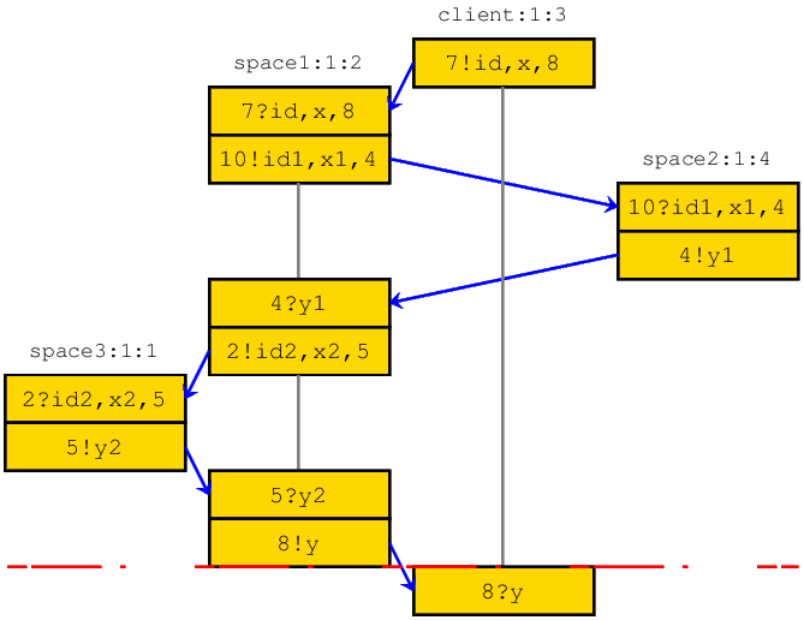

Figure 3. Correct execution: the client process obtains the requested response.

scenario 2. The parameters of the called services do not match (Figure 4). From Table 2, before test, the input parameters for the service s_1 does not match the space 1 and 2;

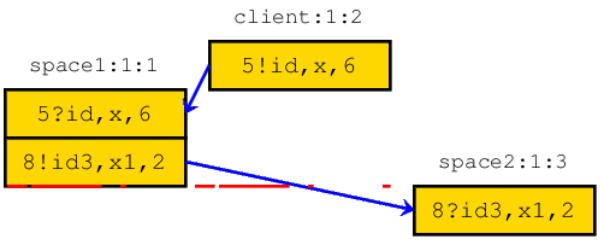

Figure 4. Not matching service: service parameters do not match. This implies the call of the client process does not execute until the end.

scenario 3. The called service is not defined (Figure 5). From Table 2, before test, s_2 service has been removed.

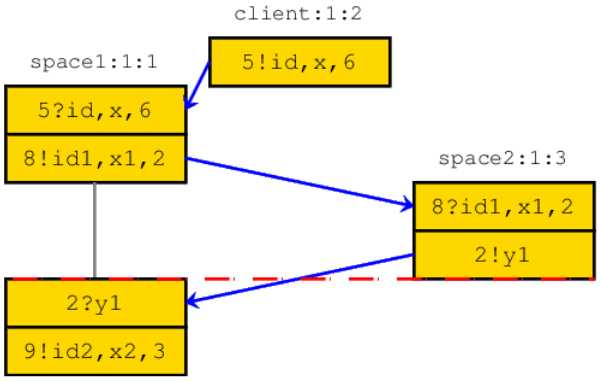

Figure 5. Undefined service: the requested process does not exist. This implies the call of the client process does not execute until the end.

Table 2. Translation of each of spaces 1, 2 and 3 to PROMELA processes.

Space	Promela Code
space 1 $s(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$ $s_1(\overline{\overline{x}}_1)\ \langle\overline{\overline{y}}_1\rangle[\alpha_1]$ $s_2(\overline{\overline{x}}_2)\ \langle\overline{\overline{y}}_2\rangle[\alpha_2]$ $s_3(\overline{\overline{x}}_3)\ \langle\overline{\overline{y}}_3\rangle$ $s_3(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space1(){** **begin:** $\alpha?s, x, p$ **if** $::(s == id) \rightarrow$ $\alpha_1!id_1, x_1, p_1;$ $p_1?y_1$ $\alpha_2!id_2, x_2, p_2;$ $p_2?y_2$ $y_3\ \ //definey_3$ $p!y$ **goto begin;** **fi** }
space 2 $s_1(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space2(){** **begin:** $\alpha_1?s, x, p$ **if** $::(s == id1) \rightarrow$ $p!y_1$ **goto begin;** **fi** }
space 3 $s_2(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space3(){** **begin:** $\alpha_2?s, x, p$ **if** $::(s == id2) \rightarrow$ $p!y_2$ **goto begin;** **fi** }

5. Technical Framework

This section presents a design and execution environment for the proposed language, an example of modeling in the environment, and the contributions of our approach. Figure 6 summarizes our validation framework.

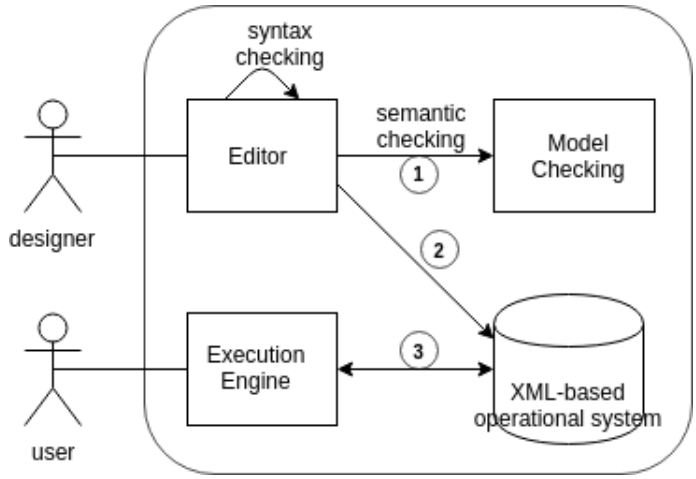

Figure 6. Validation Framework: 1. The services specified in the editor are transformed into Promela for model checking. 2. The services specified are transformed into XML for operationalization. 3. The execution engine manages the XMLs.

1. check that all service specifications are consistent or that any changes in the services are consistent.;
2. refinement of the service specification for the peer to the operational system when the point 1 is correctly done. Here the elements like the nature (RESTfull or Manually) of the services and their location are added;
3. the execution engine uses XML-based operational system to meet users needs by dynamically combining different services.

5.1. Tools

A prototype has been developed. It consists of an editor, a transformation engine and an execution engine.

5.1.1. The Editor

We have developed a DSL for the proposed language using Xtext [34,35]. The eclipse plugins obtained allows doing a syntax check when editing the services and refine the specification to the operational system for the role when the verification succeeds. The EMF-based Abstract Syntax Tree in Figure 7 makes it possible to check this syntax.

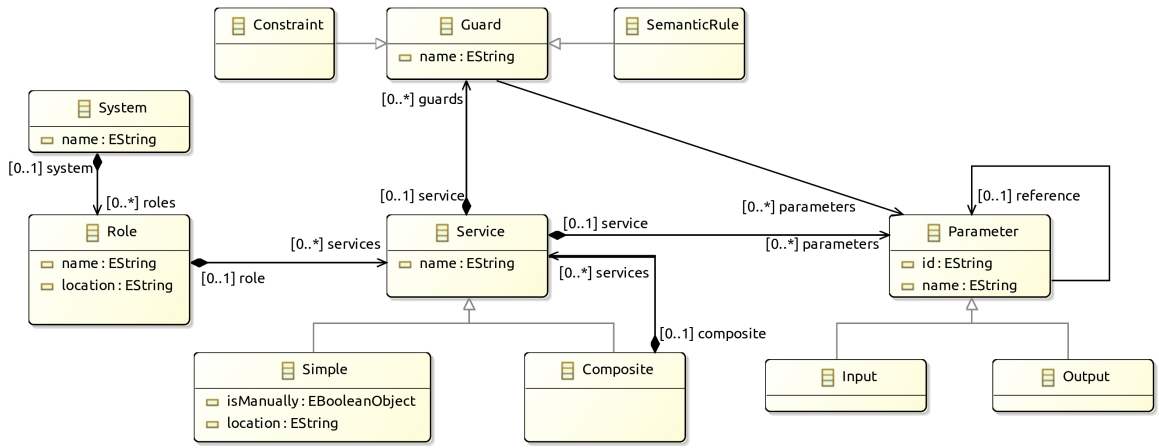

Figure 7. EMF-based Abstract Syntax Tree for the Services Specification.

5.1.2. The Transformation and Semantic Verification Engine

The transformation of Table 1 has been automated using the ATL language [36]. The source is the Abstract Syntax Tree in Figure 7 and the target is the Promela meta-model presented in document [37]. The main transformation rules are:

- **guard2Condition** rule: transforms the guards associated with the service into a condition for IF statement of promela.
- **service2ChannelAndSimpleStatement** rule: transforms simple services into channel-type variables and simple statement in a promela proctype process.
- **service2ChannelAndCompositeStatement** rule: transforms composite services into channel-type variable and composite statement such as the If statement
- **role2Process** rule: transforms a role (peer) into a promela process.
- **system2Model** rule: transforms the system composed of the set of roles into a promela model composed of all the processes.

A skeleton of the Transformation Rules Code is shown in Figure 8.

```
SpecServToProme | Promela class d | specService cla | promela.ecore | specServExample

81  rule Sytem2Model {
82      from syst: specService!System
83      to model : promela!Model (
84          name <- syst.name,
85          variables <-
86              Set {
87                  syst.roles ->  collect (e | thisModule.resolveTemp(e, 'chan1')),
88                  specService!Service.allInstances()  ->  collect(e | thisModule.resolveTemp(e, 'prim1')),
89                  specService!Service.allInstances()  ->  collect(e | thisModule.resolveTemp(e, 'prim11')),
90                  specService!Input.allInstances() -> collect(e | thisModule.resolveTemp(e, 'prim2')),
91                  specService!Output.allInstances() -> collect(e | thisModule.resolveTemp(e, 'prim3'))
92              } ->flatten() ,
93
94          processes <- syst.roles -> collect(e |thisModule.resolveTemp(e, 'process1')) ->flatten()
95
96      )
97  }
98
99  rule Role2Process{
100     from rol3 : specService!Role
101     to chan1 : promela!Channel (
102         size <- 5,
103         type <- #byte,
104         typeName <- '{mtype,mtype,chan}',
105         name <- rol3.name+'pub',
106         model <- thisModule.resolveTemp(rol3.system, 'model')
107     ),
108     process1 : promela!Process(
```

Figure 8. skeleton of transformation rules. For example, System is translated into Promela Model, Role is translated to Promela Process.

5.1.3. The Execution Engine

The execution engine consists of two parts:

- Web interfaces developed in Java using the JSF/Primefaces framework.Each role can use these interfaces to visualize the artifact contained and their instances. The web interface is shown in Figure 9. After the login of a role, it presents the services available for the role (at the top) and the service instances that are running (at the bottom).
- A communication medium that is a middleware of the Message Oriented Middleware (MOM) type because the MOMs allow interactions between application components in a loosely coupled, asynchronous and reliable framework. For this purpose, JMS (Java Message Service) was used.

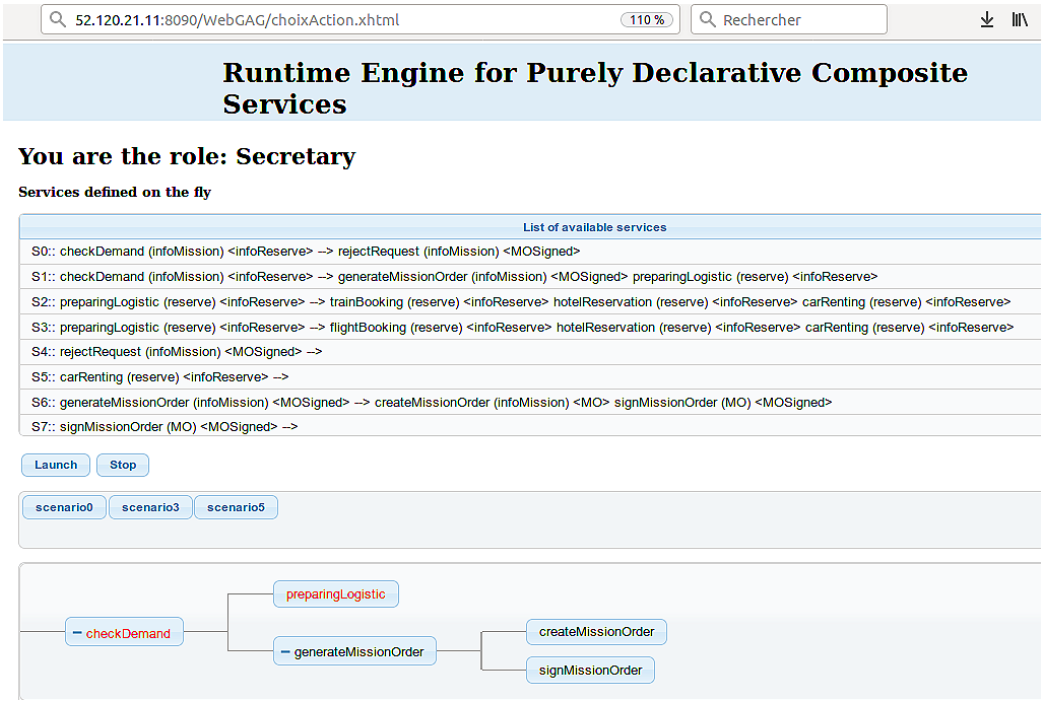

Figure 9. Main interface of the Execution Engine, the top part shows the specified services and the bottom part the instance of the running service.

In the following section, an example shows how modeling and execution are implemented.

5.2. *Case Study*

5.2.1. Case Description

Let us consider a simplified mission management system (MMS) used by an organization to organize the business travel of its employees. This case study completes the description of the example which have been presented at the beginning of this paper. It is selected firstly to show the use of the proposed language on a concrete example. Secondly, to validate the environment which is put in place. In this regard, this example involves three actors: Employee, Secretary, Head of Department (HOD), and four services provided by external structures. The employee requests a mission order by filling a form which is submitted to the Secretary. The Secretary checks the validity of the demand. If it is approved, the form is transmitted to the HOD while the logistic is prepared. The latter action is made up with external services. It uses *Flight Booking* or *Train Booking* service, *Hotel Reservation* service and *Car Rent* service. Figure 10 shows the overview of the example.

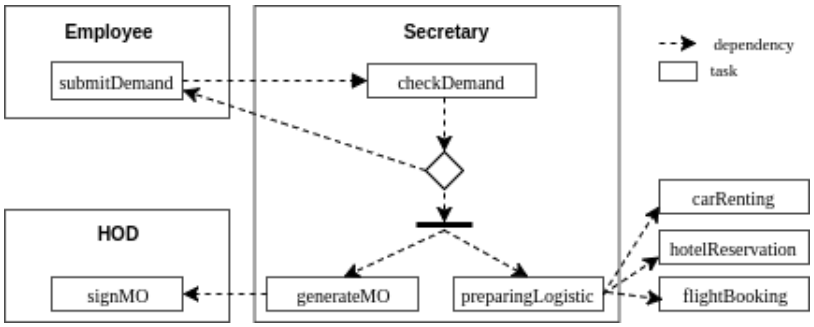

Figure 10. Structure of the Mission Management System (MMS).

5.2.2. Modelling in the Environment and Verification

The current version of the editor is textual. It offers the possibility of editing and checking the syntax of the services.

Figure 11 shows the edition of the rules associated with the example of the previous section. Each role is represented, in each role we have the associated services. For example in the role Employee we have the *submitDemand* service.

Figure 11. Modeling in the Editor: The variables, the external services and the defined services of each role are presented.

Once the modeling is completed, the code is verified and in case of success, the transformation engine translate it into Promela for semantic verification. Then, the editor generates the XML model which is loaded in the execution engine.

In summary

- each peer contains the editor and all the service specification of the system because the verification is done over the entire specification;
- on a peer, the operational services generated are only those of the roles;
- an update of the service specification is reflected in the different peer after the verification has been successful.

5.2.3. Execution

The architectural pattern induced by the model is a P2P style implemented on the top of the chosen MOM as shown in Figure 12.

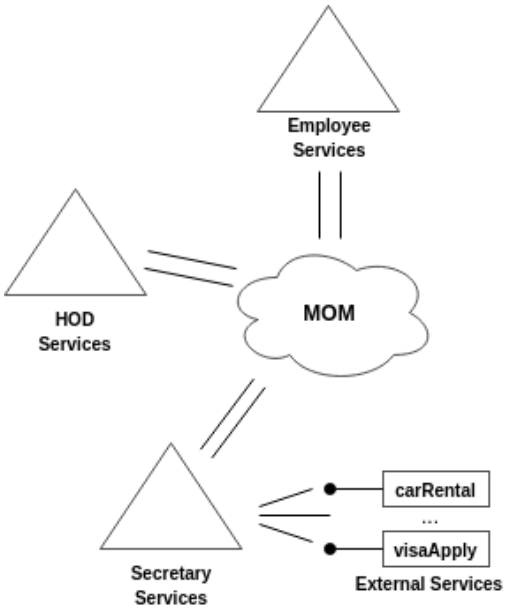

Figure 12. The architectural style of the execution engine is P2P. The employee, the HOD and the Secretary exchange data via MOM-type middleware.

The execution engine has a visual aspect (web interface) and a backend that implements the rules of Section 3.3 and is based on the MOM.

The debugging aspect of this work is the possibility offered to the users to observe the evolution of the service instances as shown in Figure 13 where an instance of the *checkDemand* service is presented. The refined nodes as well as related information can be viewed. We can request each node of the *checkDemand* service for the details of these informations.

The black nodes are the services already treated and the red nodes are the pending services waiting for an event or information to continue their execution.

The schema of a service can be updated in the editor and reload in the execution engine without stopping the running processes. This modification will take effect next time the given service is chosen to be applied.

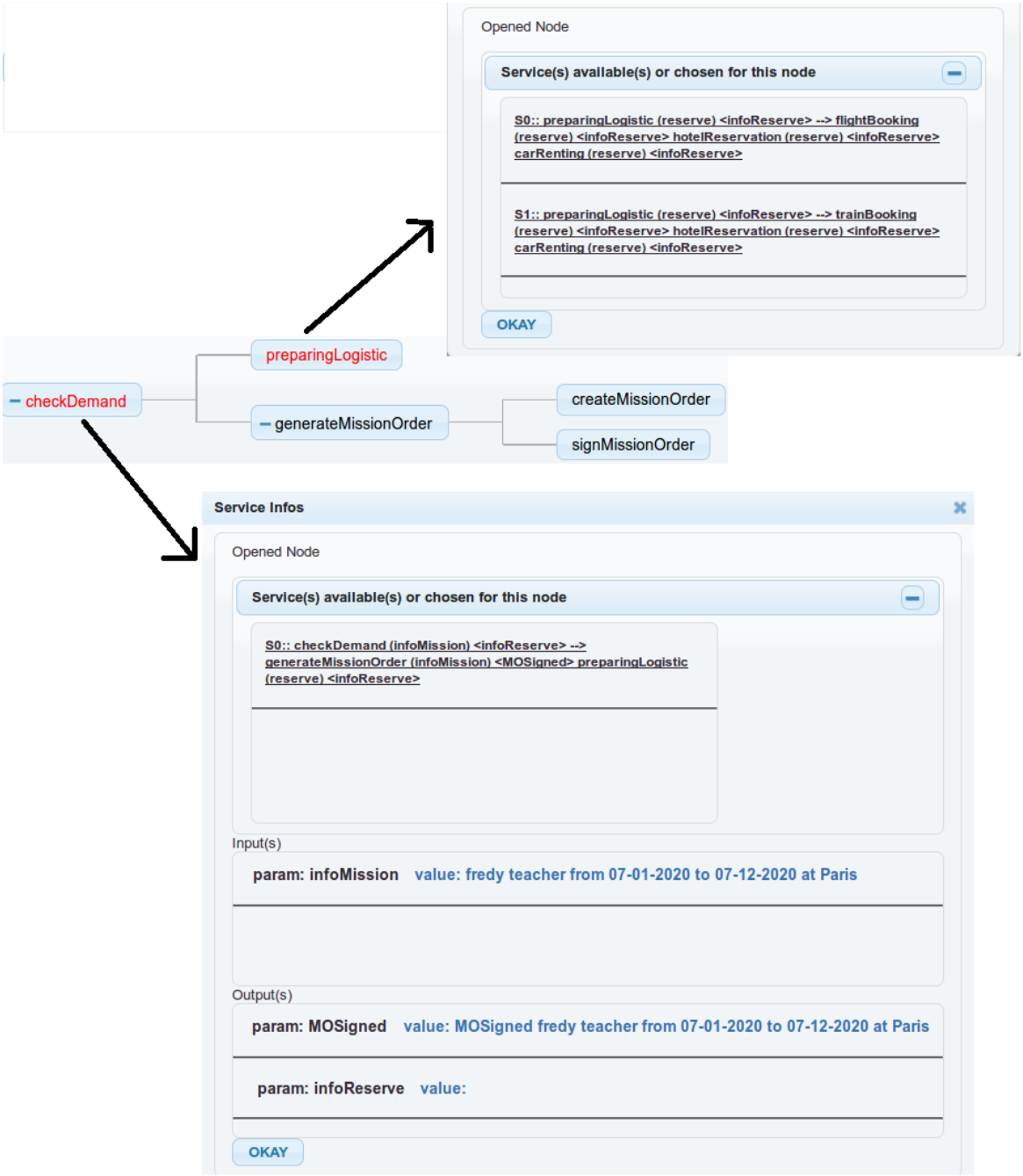

Figure 13. A runtime instance of the CheckDemand service. At this stage, preparingLogistic proposes to choose between S_0 and S_1 and the checkDemand service is still waiting for the values of the infoReserve variable.

6. Discussion

Several studies have been interested in the composition of services and more recently declarative languages have been proposed in order to provide more flexibility and adaptation during the execution of services. In this part, The choreography (peer to peer architecture) as well as the declarative paradigm for the composition of services are discussed. We end by highlighting the data-driven in the composition of services.

6.1. Architecture Style

As an alternative to traditional languages and notations for service compositions such as BPEL [10] and BPMN [38], other languages, such as JOpera [39] and Jolie [40] have been proposed. Nevertheless, they favor the orchestration, a single point of view for the composition of service. Being of type orchestration, they do not allow to consider the collaborations of peer to peer. Our language is instead of a choreographic type.

Few approaches focus on choreography for compositional modeling. Inspired by the WS-CDL [12] standards, over the past decade, choreography has been studied to support a new programming paradigm called choreographic programming [41,42]. In choreographic programming, the programmer uses choreography to program the service systems, and then a compiler is used to automatically generate compliant implementations. This gives an accuracy method by construction, guaranteeing important properties such as freedom of blocking and the absence of communication errors. This aspect of pre-deployment verification is important in the collaboration between services, which is why in our language we transform our specification into promela. Nevertheless, the choregraphy programming allows a global description of the interactions between the different sites. In other words, the interactions are known a priori what we do not want. It is difficult to change running choreography when services are added/removed as needed by the designer.

The approaches such as BPEL4Chor [43] tend to add the choreography in orchestration style. Nevertheless, it relies on BPEL, which has been criticized for its rigid character.

6.2. Declarative Approaches

Many languages [39,40,44] have been proposed. Although easier [45] to use and often more expressive than BPEL and BPMN [38], they do not deviate from the imperative paradigm and, therefore, share with them the same limitations that motivated our work.

To provide flexibility by change [28], the declarative languages have emerged. Among them, declarative approaches such that Declare [46]. Declare is closest to our work considering its abstraction level. In Declare, service compositions are defined *as a set of actions and the constraints that affect them*. Actions and constraints are modeled, while constraints have a formal semantics given in linear temporal logic (LTL).

There are several differences between Declare and our approach: First, Declare focuses on the modeling of service orchestrations to support auditing and monitoring while our language focuses on the modeling of choreographies. In addition, Declare focuses on specification and verification and does not provide specific mechanisms for handling run-time failures.

GO-BPMN [47] is another declarative language, designed as a goal-oriented extension for traditional BPMN. In GO-BPMN, business processes are defined as a hierarchy of goals and sub-goals. Multiple BPMN plans are attached to the "leaf" goals. When executed, they achieve the associated goal. These plans can be alternative or they can be explicitly associated with specific conditions through guard expressions based on the context of execution. Although this approach also tries to separate the declarative statements from the way they can be accomplished, the alternative plans to achieve a goal must be explicitly designed by the service architect and are explicitly attached to their goals. The engine does not automatically decide how the plans are built or replaced; it just chooses between the given options for each specific goal, and it does it at service invocation time. GO-BPMN relies on BPMN known as being rigid to change. In our approach for an instance of a given composite, we can modify the structure of the services not applied which makes it possible to change its way of executing itself.

SelfMotion [45] is a declarative language that uses Abstract Actions which specify the primitive operations that can be executed to achieve the goal; Concrete Actions, one or more for each abstract action, which map them to the executable snippets that implement them. As our approach, it manages the adaptation of composite services at runtime. Nevertheless, it does not check in order to prevent some runtime malfunction as it focuses more on implementation.

Concerning SELF-SERV [48], A major outcome of the project has been a prototype system in which Web services are declaratively composed. In SELF-SERV, the process model underlying a composite service is specified as a statechart whose states are labeled with invocations to Web services, and whose transitions are labeled with events, conditions, and variable assignment operations. Statecharts possess a formal semantic and offers most of the constructs found in contemporary process modeling languages (sequence, branching, structured loops, concurrent threads, inter-thread synchronization, etc.). This ensures that the service composition mechanisms and orchestration techniques developed within the SELF-SERV project can be adapted to other process modeling languages for Web Services such as BPEL4WS and WSCI. Although operating in a peer-to-peer environment and explicitly taking into account roles that is one of the characteristics of our approach, it based its design on imperative paradigms thus inheriting all its shortcomings.

6.3. Data-Driven

The data are one of the important points, they allow to specify how the outputs derive from the inputs [21]. The operationnalization of the proposed language implies a data-driven approach since the models issued from the here defined language are piloted by the data. This vision is close to Active XML [49] where the remote services are inserted in XML documents. These services, so called intentional data, are enabled when loading the XML document. In our approach, the artifacts contain the concrete data which represents the already enacted service and the intentional data that represent the pending services which could be applied. Nevertheless Active XML is dedicated to data integration rather than service composition.

Mashups [50] assemble the services coming from many sources in one web page. Some disadvantages of the mashups are:

1. The SOA developers usually need to spend major effort to master many SOA technologies, e.g., BPEL, WSDL, SCA (Service Component Architecture), as well as tools, e.g., design-time IDE tools, and runtime middleware servers (SCA server or BPEL server). Most of these tools require major investment on hardware and software infrastructure;

2. These technologies cannot support service composition's customization on the fly because the process of service composition (design, development and testing) is usually conducted in IDE tool first according to customer requirements, and then deployed on the runtime server. After deployment, the composition logic is not easy to customize according to the changes of composite service requirements, as this involves to come back at the earlier phases of the development process.

7. Conclusions and Future Works

In this paper, we proposed a rule-based approach to the definition of composite services in order to support adaptability and flexibility by change at runtime. The formalism used is that of attributed grammars extended with guards where the structure of a service is represented in a declarative purely manner. This approach focuses on the description of the execution schema of the composition while the process-based approaches describe the execution path of the composition. Formal description and verification ensures the correct composition of web services. It also ensures that system works as it is expected. The syntactic elements as well as an operational semantics of this language are described in pi-calculus formalism. The operational semantics makes it possible to highlight the dynamism of the services defined through out the asynchronous ports created at runtime. A verification framework which consists of translating the defined services into promela is proposed. This is done in order to check the consistency of the specified services and their ending. The validation framework allows the evaluation by proposing the tools. It includes an editor, a checker and an engine to execute the specification.

The relevance of our approach could be appreciated especially in the case of composing a very large number of services, or in the case of services that take time to execute. Many properties are highlighted in the model presented in this paper:

- **Declarative:** the language described is based on the rules that describe a service by specifying its name, input and output parameters, semantic rules (describing the correspondence between attributes) and services to solve it.
- **Abstract Specification:** The intentional definition of services allows a late concretization of the services thus favoring a weak culprit with the underlying technology and an adaptation (updating the rules) of the service even during its execution.
- **Connectivity (Asynchronous):** Sends and receives asynchronous messages for the creation and refinement of services. Indeed, a service is implemented in an execution space with, among other things, a public port for communication. Dynamically created ports work together to refine instances of services.
- **Adaptation and Flexibility:** Services can be defined at any time, verified and integrated into the execution spaces.
- **Distribution:** Services are distributed in different locations (user space or service provider).
- **Data driven:** The services parameters guide for the choice of the services. Depending on these parameters, the user or execution engine does the choice of the service to instantiate.

Future work could be associated with the concerns of the semantic web with respect to the problems of automatic search for missing services and the automatic generation of composite services from user requests. To the language thus described could be added constraints that could guide the automatic composition of services and the specification of the non-functional requirements that would favor the construction of composite services based on user requests.

Author Contributions: Conceptualization, W.K.K. and G.-E.K.; Methodology, W.K.K. and G.-E.K.; software, W.K.K.; formal analysis, W.K.K. and G.-E.K.; writing–original draft preparation, W.K.; writing–review and editing, G.-E.K.; supervision, C.T.; funding acquisition, G.-E.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: This research was supported by FUCHSIA project-team of LIRIMA funded by Inria and the CETIC's Programme of the World Bank.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

ATL	Atlas Transformation Language
BPEL	Business Process Execution Language
BPML	Business Process Modeling Language
GAG	Guarded Attribute Grammars
LHS	Left-Hand Side
REST	REpresentational State Transfer
RHS	Right-Hand Side
SELF-SERV	compoSing wEb accessibLe inFormation & buSiness sERVices
SOAP	Simple Object Access Protocol
SPIN	Simple Promela INterpreter
BPEL4WS	Business Process Execution Language for Web Services
WS-CDL	Web Services Choreography Description Language
WSDL	Web Services Description Language
WSCI	Web Service Choreography Interface
PROMELA	PROcess Meta LAnguage

References

1. Rao, J.; Su, X. A survey of automated web service composition methods. In Proceedings of the International Workshop on Semantic Web Services and Web Process Composition, San Diego, CA, USA, 6 July 2004; pp. 43–54.
2. Sheng, Q.Z.; Qiao, X.; Vasilakos, A.V.; Szabo, C.; Bourne, S.; Xu, X. Web services composition: A decade's overview. *Inf. Sci.* **2014**, *280*, 218–238. [CrossRef]
3. Barakat, L.; Miles, S.; Luck, M. Adaptive composition in dynamic service environments. *Future Gener. Comput. Syst.* **2018**, *80*, 215–228. [CrossRef]

4. Nagl, C.; Rosenberg, F.; Dustdar, S. VIDRE–A Distributed Service-Oriented Business Rule Engine based on RuleML. In Proceedings of the 2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), Hong Kong, China, 16–20 October 2006; pp. 35–44.
5. Kamada, A.; Mendes, M. Business rules in a service development and execution environment. In Proceedings of the 2007 International Symposium on Communications and Information Technologies, Sydney, Australia, 17–19 October 2007; pp. 1366–1371.
6. Papazoglou, M.P.; Benbernou, S.; Andrikopoulos, V. On the evolution of services. *IEEE Trans. Softw. Eng.* **2012**, *3*, 609–628.
7. Weigand, H.; van den Heuvel, W.J.; Hiel, M. Rule-based service composition and service-oriented business rule management. In Proceedings of the International Workshop on Regulations Modelling and Deployment (ReMoD'08), Montpellier, France, 17 June 2008.
8. Badouel, E.; Hélouët, L.; Kouamou, G.E.; Morvan, C.; Fondze J.N. Active workspaces: Distributed collaborative systems based on guarded attribute grammars. *ACM SIGAPP Appl. Comput. Rev.* **2015**, *15*, 6–34. [CrossRef]
9. Sangiorgi, D.; Walker, D. *The pi-Calculus: A Theory of Mobile Processes*; Cambridge University Press: Cambridge, UK, 2003.
10. Jordan, D.; Evdemon, J.; Alves, A.; Arkin, A.; Askary, S.; Bloch, B.; Curbera,F.; Ford, M.; Goland, Y.; Guízar, A.; et al. Web services business process execution language version 2.0. *OASIS stand.* **2007**, *11*, 5.
11. Dijkman, R.; Hofstetter, J.; Koehler, J. *Business Process Model and Notation*; Springer: Berlin, Germany, 2011, pp. 10–12.
12. Kavantzas, N.; Burdett, D.; Ritzinger, G.; Lafon, Y. *Web Services Choreography Description Language Version 1.0, W3c Candidate Recommendation*; Technical Report; W3C: Cambridge, MA, USA, November 2005.
13. Assaf, A.; Intalio, A.A.; Intalio, S.A.; Fordin, S.; Sap, W.J.; Kawaguchi, K.; Orchard, D. Web Service Choreography Interface 1.0. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.879&rep=rep1&type=pdf (accessed on 25 January 2020).
14. Sabraoui, A.; Ettalbi, A.; El Koutbi, M.; En-Nouaary, A. Towards an UML profile for web service composition based on behavioral descriptions. *J. Softw. Eng. Appl.* **2012**, *5*, 711. [CrossRef]
15. Newcomer, E. *Understanding Web Services: XML, Wsdl, Soap, and UDDI*; AddisonWesley Professional: Boston, MA, USA, 2002.
16. Papazoglou, M.P.; Van Den Heuvel, W.J. Service oriented architectures: Approaches, technologies and research issues. *VLDB j.* **2007**, *16*, 389–415. [CrossRef]
17. Yang, H.; Zhao, X.; Qiu, Z.; Pu, G.; Wang, S. A formal model forweb service choreography description language (WS-CDL). In Proceedings of the 2006 IEEE International Conference on Web Services (ICWS'06), Chicago, IL, USA, 18–22 September 2006; pp. 893–894.
18. Martin, D.; Paolucci, M.; McIlraith, S.; Burstein, M.; McDermott, D.; McGuinness, D.; Parsia, B.; Payne, T.; Sabou, M.; Solanki, M.; et al. Bringing semantics to web services: The OWL-S approach. In Proceedings of the International Workshop on Semantic Web Services and Web Process Composition, San Diego, CA, USA, 6 July 2004; pp. 26–42.
19. Berners-Lee, T.; Connolly, D.; Kagal, L.; Scharf, Y.; Hendler, J. N3logic: A logical framework for the world wide web. *Theory Pract. L. Program.* **2008**, *8*, 249–269. [CrossRef]
20. Rosenberg, F.; Dustdar, S. Business rules integration in BPEL-a service-oriented approach. In Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05), Munich, Germany, 19–22 July 2005; pp. 476–479.
21. Yao, Y.; Chen, H. A rule-based web service composition approach. In Proceedings of the 2010 Sixth International Conference on Autonomic and Autonomous Systems, Cancun, Mexico, 7–13 March 2010; pp. 150–155.
22. Zhu, Y.; Huang, Z.; Zhou, H. Modeling and verification of web services composition based on model transformation. *Softw. Pract. Exp.* **2017**, *47*, 709–730. [CrossRef]
23. Abouzaid, F.; Mullins, J. Model-checking web services orchestrations using bp-calculus. *Electron. Notes Theor. Comput. Sci.* **2009**, *255*, 3–21. [CrossRef]
24. Bianculli, D.; Ghezzi, C.; Spoletini, P. A model checking approach to verify BPEL4WS workflows. In Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA '07), Newport Beach, CA, USA, 19–20 June 2007; pp. 13–20.

25. Paschke, A.; Kozlenkov, A. A Rule-Based Middleware for Business Process Execution. Available online: https://vsis-www.informatik.uni-hamburg.de/events/mas2/paschke.pdf (accessed on 23 January 2020).
26. Orriëns, B.; Yang, J.; Papazoglou, M.P. A framework for business rule driven service composition. In *International Workshop on Technologies for E-Services*; Springer: Berlin/ Heidelberg, Germany, 2003; pp. 14–27.
27. Charfi, A.; Mezini, M. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web* **2007**, *10*, 309–344. [CrossRef]
28. Schonenberg, H.; Mans, R.; Russell, N.; Mulyar, N.; van der Aalst, W.M. Towards a Taxonomy of Process Flexibility. In Proceedings of the Forum at the CAiSE'08 conference, Montpellier, France, 18–20 June 2008; pp. 81–84.
29. Knuth, D.E. The genesis of attribute grammars. In *Attribute Grammars and Their Applications*; Springer: Berlin/Heidelberg, Germany, 1990; pp. 1–12.
30. Kungne, W.K.; Kouamou, G.E.; Tangha, C. Introducing an Artifact-driven language for Service Composition. In Proceedings of the ArabWIC 6th Annual International Conference Research Track, Rabat, Morocco, 6–8 March 2019; pp. 1–6.
31. Holzmann Gerard, J. *SPIN Model Checker: The Primer and Reference Manual*; Addison Wesley: Boston, MA, USA, 2003.
32. Wu, P. Interpreting π-calculus with Spin/Promela. *Comput. Sci.* **2003**, *8*, 9.
33. Song, H.; Compton, K.J. Verifying π-calculus processes by Promela translation. Available online: https://www.researchgate.net/profile/Kevin_Compton/publication/244354070_Verifying_-calculus_Processes_by_Promela_Translation/links/00b4953232bc753d31000000.pdf (accessed on 23 January 2020).
34. Eysholdt, M.; Behrens, H. Xtext: Implement your language faster than the quick and dirty way. In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages And Applications Companion, Reno/Tahoe, NE, USA, 17–21 October 2010; pp. 307–309.
35. Bettini, L. *Implementing Domain-Specific Languages with Xtext and Xtend*; Packt Publishing: Birmingham, UK, 2016.
36. Jouault, F.; Allilaire, F.; Bézivin, J.; Kurtev, I. ATL: A model transformation tool. *Sci. Comput. Program.* **2008**, *72*, 31–39. [CrossRef]
37. Gentile, U. A Model-driven Approach for the Automatic Generation of System-Level Test Cases. Ph.D. Thesis, University of Naples Federico II, Naples, Italy, 2016.
38. White, S.A. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*; Future Strategies Inc.: Toronto, ON, Canada, 2008.
39. Pautasso, C.; Alonso, G. JOpera: a toolkit for efficient visual composition of web services. *Int. J. Electron. Commer.* **2005**, *9*, 107–141. [CrossRef]
40. Montesi, F.; Guidi, C.; Lucchi, R.; Zavattaro, G. Jolie: A java orchestration language interpreter engine. *Electron. Notes Theor. Comput. Sci.* **2007**, *181*, 19–33. [CrossRef]
41. Thönes, J. Microservices. *IEEE softw.* **2015**, *32*, 116–116. [CrossRef]
42. Shadija, D.; Rezai, M.; Hill, R. Towards an understanding of microservices. In Proceedings of the 2017 23rd International Conference on Automation and Computing (ICAC), Huddersfield, UK, 7–8 September 2017; pp. 1–6.
43. Decker, G.; Kopp, O.; Leymann, F.; Weske, M. BPEL4Chor: Extending BPEL for modeling choreographies. In Proceedings of the IEEE International Conference on Web Services (ICWS 2007), Salt Lake City, UT, USA, 9–13 July 2007; pp. 296–303.
44. Kitchin, D.; Quark, A.; Cook, W.; Misra, J. The Orc programming language. In *Formal Techniques for Distributed Systems*; Springer: Berlin/Heidelberg, Germany, 2009, pp. 1–25.
45. Cugola, G.; Ghezzi, C.; Pinto, L.S.; Tamburrelli, G. Selfmotion: A declarative approach for adaptive service-oriented mobile applications. *J. Syst. Softw.* **2014**, *92*, 32–44. [CrossRef]
46. Montali, M.; Pesic, M.; van der Aalst, W.M.; Chesani, F.; Mello, P.; Storari, S. Declarative specification and verification of service choreographiess. *ACM Trans. Web (TWEB).* **2010**, *4*, 3. [CrossRef]
47. Greenwood, D.; Rimassa, G. Autonomic goal-oriented business process management. In Proceedings of the Third International Conference on Autonomic and Autonomous Systems (ICAS'07), Athens, Greece, 19–25 June 2007; p. 43.

48. Benatallah, B.; Sheng, Q.Z.; Dumas, M. The self-serv environment for web services composition. *IEEE Internet Comput.* **2003**, *7*, 40–48. [CrossRef]
49. Abiteboul, S.; Benjelloun, O.; and Milo, T. The Active XML project: An overview. *VLDB J.* **2008**, *17*, 1019–1040. [CrossRef]
50. Garriga, M.; Mateos, C.; Flores, A.; Cechich, A.; Zunino, A. RESTful service composition at a glance: A survey. *J. Netw. Comput. Appl.* **2016**, *60*, 32–53. [CrossRef]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).