Z-Based Agents for Service Oriented Computing

Ioan Alfred Letia[1], Anca Marginean[1], and Adrian Groza[1]

Technical University of Cluj-Napoca
Department of Computer Science
Baritiu 28, RO-400391 Cluj-Napoca, Romania
letia@cs.utcluj.ro

Abstract. Ensuring reliability and adaptability of web services represents one of the main prerequisites for a larger acceptance of web services technology. We present an agent based framework to model the global behavior of atomic e-service and their composition using Z. We consider failures associated with web services and we try to handle runtime exceptions through formal methods for specification and verification of a composite service. In addition, our framework enforces the quality of services, in terms of answer time, by providing Z-agents responsible for these aspects.

1 Introduction

Service oriented computing [1] involves loosely coupled activities among two or more business partners. Orchestration describes the way the fine-grained services can interact with each other at the message level to provide more coarse-grained business services which can be incorporated into workflows. The process oriented languages used in orchestration assume that the services combination is predefined, which deeply affects business process reconfiguration.

We handle the dynamic nature of service composition using a formal specification that integrates process oriented paradigms with ontological knowledge. Interleaving the execution phase with synthesis allows us to handle the predicted and unpredicted situations that may arise as a result of services enactments. On the one hand, the formal specification of the collaborating participants validates and guarantees the correct execution of the composite service. On the other hand, the transactional states within the composite service are flexibly manipulated in order to achieve fault tolerance and robustness [2] during service enactment.

Even though for simple service specification there is a formal approach included in WSDL version 2.0 [3], there has been essentially no formal work to understand the relationship between the global properties of the composite service and the local properties of its atomic components [4]. Out contribution consists in introducing a framework for modeling and specifying the global behavior of e-service composition through the Z language, in a multi-agent context.

2 An Agent Architecture for handling services

Effective composition of web services relies on concepts thoroughly studied in distributed computing, artificial intelligence and multiagent systems. Web services are closely related to the agent programming paradigm. The definition of the web services architecture states that a web service is an abstract notion that must be implemented by a concrete agent [5]. Even so, web services do not currently assign any large role to agents and their interactions are still limited to simple request-response exchanges [6]. MAS mediating web services introduces a new kind of architecture, in which the communication patterns between agents representing a service can be considerably more varied and complex.

A MAS approach for web services enhances their capability of dealing with dynamic nature of environment and requirements. In our approach, the Z language is used by the agents for representing domain data, state model or current tasks. The Zeta-agent has the central role in the composition process, deciding the services that will be included and their enactment order. According to the Z-model, Zeta-agent recommends the next actions for each new state of the composed service and updates the current state based on responses of the enacted web services. If more than one service can be used for the current task, a reliable agent chooses the proper one as regard to some quality criteria. A type-checker agent decides how the operations are invoked.

2.1 Dealing with failures

The composite service must deal with both deterministic and nondeterministic failures. A deterministic failure occurs when an atomic service returns a negative answer which makes the composite service to deviate from the initial plan. When the service does not respond within a time limit a nondeterministic failure has arisen, denoted by NF (Network Failure). Furthermore, handling failures must meet time constraints. The client provides a Δ_{max} time limit within which the composite task must be accomplished. This time is split according to some criteria and the milestones t_i are attached to each subtask. When a subtask is accomplished, the remaining milestones are adjusted. The reliability of the composed service is ensured by monitoring its state and re-synthesizing it when facing a failure.

The scenario used for testing our ideas is inspired from common medical activities. A physician initiates a consultation for a patient *Purgon* by sending the task *Consult*(*Purgon*) to a composition agent. Firstly, the generated composite service should check for the patient's insurance. It can exist more than one web services representing insurance houses, which may be simultaneously interrogated. If the patient has an insurance, the patient history service is asked to provide medical profile of the patient. Next, the physician consults the patient and he writes a prescription, which is sent to the the hospital secretary service. The secretary service updates the patient profile and it also requests the insurance house to pay for the consultation. Deterministic failures can appear on each state, for example if the patient does not have yet an insurance. The

medical profile is not indispensable, therefore the consultation can begin after its allocated time has expired, even though the medical history service has not yet responded (nondeterministic failure).

3 Service specification

This section covers the structural and behavioral specifications of the atomic services in Z language and it also identifies the knowledge involved in the composition process, encapsulated as business rules. *Operational reasoning*, including composition approaches in BPEL style, are fallible due to their limited adaptability restricted to some expected course of actions. In contrast, *the formal reasoning* views a composite service as a formula of which properties can be rigorously checked. The testing of composite service through execution, common to operational reasoning, is not sufficient to ensure its correctness in the presence of a large number of participating services and of the nondeterminism arising from the behavior of these services.

3.1 Structural Specification

The formal structural model is generated from the WSDL specification, translated by an automated parser to Z language [7] which defines a Z generic type for each WSDL component. The Z *Component* type represents the collection of all these generic types for operations, messages and their components (figure 1).

$$Component ::= element\langle\langle Element\rangle\rangle \mid part\langle\langle Part\rangle\rangle \mid message\langle\langle Message\rangle\rangle \mid operation\langle\langle Operation\rangle\rangle \mid messageref\langle\langle MessageReference\rangle\rangle$$

Operation
Identifier
name : *QName*
type : *OperationType*
messageRefs : $\mathrm{seq}\,\mathbb{N}$

MessageReference
Identifier
message : $\mathbb{N}$
direction : *Direction*

Fig. 1. Main types for WSDL description

The formal structural description makes possible the verification of referential integrity for the concrete service model. Instances of these generic types are used in type checking and behavioral specification of the service, and also for the effective enactment of the services.

3.2 Behavioral Specification

Inspired by flow composition, the behavioral specification augments the service specification with states modeling elements based on structural specification and additional knowledge contained by the WSDL-S descriptions. For each WSDL operation, a state transition is defined specifying its input values, the operation and the global task that can be accomplished. At this level, an operation op! is specified by its name and sequence of parameters corresponding to the elements of involved WSDL messages. This specification can be considered as a procedural translation of the structural one. For example, the transition *Pay*1 achieves the *pay* task by executing WSDL operation *Pay*1 with parameters *patient*? and *fee*?.

*Pay*1

$fee? : \mathbb{N}$

$patient? : QName$

$task! : TASK$

$op! : OPER$

$op! = op1(\!|\ name == pay1,$
$\quad param == \langle string(\!|\ value == patient?\ |\!),$
$\quad\quad nat(\!|\ value == fee?\ |\!)\rangle\ |\!)$

$task! = pay$

$OPER ::= op1\langle\langle Operation1\rangle\rangle$

*Operation*1

$name : QName$

$param : \mathrm{seq}\ ParamType$

There are no specified preconditions for transitions at this level. More behavioral information is added from the business rules at a centralized level. Inside a service community all the transitions with the same task as output are supposed to have the same final functionality from a service external viewpoint.

3.3 Business Rules

Current workflow technology is often too rigid [1], meaning that the agents have limited possibilities to reason on the information provided by the current WSDL specifications. Aiming to improve the adaptability and expressiveness, we consider the composed service as a business process and we propose the use of five different types of business rules together with the formal descriptions of services, both structural and behavioral.

Domain task coordination are structural rules that define a task. In order to have a shared representation of the tasks and their associated messages, we consider a domain ontology having the following structure

- concepts: *task*, *workflow task*, *messages*. If the composed service needs the human intervention, then the workflow task is used. There can be simple and composed tasks, for the latter being necessary to achieve some other tasks.
- object properties: *subtask*, *precedes: task* $\rightarrow$ *task*, defining execution constraints between tasks. The property *hasMessage: task* $\rightarrow$ *Message* allows defining the messages associated to a simple task.

– datatype properties: *typeOfTask* and *typeOfMessage*. We model two types of tasks: $typeT \in \{informative, operational\}$ where $typeOfTask : task \rightarrow typeT$. The operational tasks imply a change at the data level of the service (as in *updateHistory*), while the informative ones (*getHistory*) are used only to get some information thus being possible to be executed more than once, even in parallel when more than one service provides operations for them[1]. Regarding the type of message property $typeOfMessage : message \rightarrow typeM$, there are two categories $typeM \in \{input, output\}$.

Querying the domain ontology provides for structural rules on tasks and execution constraints between them. Based on *precedes* and *substask* relations, one can identify different control patterns for subtasks[2]. Considering our scenario, the first two rules of the following schema represent a sequential pattern, whilst the last one is a parallel pattern. Having a hierarchical decomposition of tasks allows dealing with services of different granularity.

$$
\begin{array}{|l}
rules : \mathrm{seq}\,\mathbb{P}\, TASK \leftrightarrow TASK \\
\hline
rules = \{ \\
\quad \langle \{startConsult\}, \{endConsult\} \rangle \mapsto consult, \\
\quad \langle \{checkInsurance\}, \{getHistory\} \rangle \mapsto startConsult, \\
\quad \langle \{payConsultation, updateHistory\} \rangle \mapsto endConsult \}
\end{array}
$$

Operation constraints express at a global level the constraints of the task transitions through preconditions and effects. A task transition includes the task accomplished, the concrete operations from all known services that are able to achieve the task, preconditions, and effects expressed in terms of process and domain variables. *Process variables* are used to monitor the composition process, while the *domain variables* characterize the state of the composed service. The preconditions include both process and domain variables, whilst the effects are referring only to the domain variables.

In the example from figure 2, the *SimplePayGYes* transition accomplishes the *pay* task and it is possible only if the *pay* task has not been achieved yet (process variable condition) and the patient has insurance (domain variable condition). The effect will be the change of the domain variable *payConsultation*. The concrete operations are extracted from the behavioral specification of atomic services (in the below example, from the *Pay1* transition) and they are added to the sequence of operations in the *OpPayGYes* component of the *PayGYes* task transition. We have to observe that for the same task *pay* there can be more services providing achieving operations, and all these operations are included in *OpPayGYes*. The values of the process variables are not modified by transitions accomplishing a task as it can be seen in *HistoryComposedS* schema.

[1] A task without a *typeOfTask* property becomes operational if one of its subtasks is operational.

[2] These ones represent ontological control pattern. A different type of control patterns appears when the composite service handles preconditions and effects for each operation provided by the services.

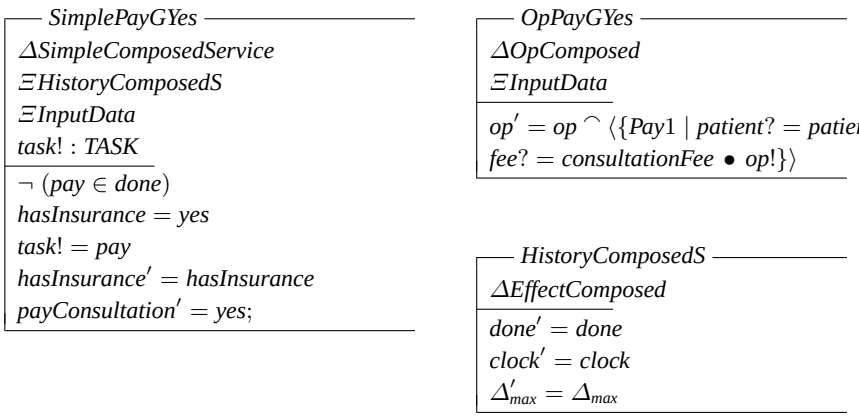

Fig. 2. *Pay* operation constraints

The task transition for the *pay* task is the result of merging these three schemes $PayGYes == OpPayGYes \wedge SimplePayGYes \wedge HistoryComposedS$. Due to the nondeterministic nature of the composite service, task transitions must catch all the possible situations. Therefore, more transitions are defined for the action of paying, one for the case the transaction was successfully done and one for the opposite case. When generating a plan, both of them are considered, but only one can be included: $PayG == PayGYes \vee PayGNo$.

Message translation rules define the changes on the process and domain variables according to the response messages of the enacted services. Based on the message rules, at every step of the composition, a new state is computed.

Business entity constraints define input requirements for the evolution of variables through the entire composition process (the total cost of the composed service, time limit constraint Δ_{max}). In the formal specification they are expressed as axioms | $total\_price < 10$ or as restrictions of type definitions. Therefore, when animating the Z formal model their truth values are verified in all transitions.

Time constraints are rules that manage instances of time allocated to each task and they influence the control flow of the composite service. Their role is to identify failures and to exclude the operations which have generated the nondeterministic ones from the re-planning process.

4 Z-Oriented Agents

In our framework we deal with a "community of web services" [8]. Each service is represented by an agent in the OAA community. The Z-based agents interact with Zeta tool[3] in order to define, verify, and animate the composite web service.

[3] http://uebb.cs.tu-berlin.de/zeta/

Composition is done in a centralized manner, having the Zeta agent as the main orchestrator. The composed service is specified by the *ComposedService* type having as main components the process variables describing the computation process included in *EffectComposed* and *OpComposed* schemes, respectively the *InputData* and the *SimpleComposedService* for the domain specific variables.

ComposedService

InputData
SimpleComposedService
EffectComposed
OpComposed

EffectComposed

$done : \mathbb{P}\, TASK$
$clock : \mathbb{N}$
$\Delta_{max} : \mathbb{N}$

$clock \leq \Delta_{max}$

OpComposed

$op : \text{seq}(\mathbb{P}\, OPER)$

4.1 General algorithm for composition

The composition process (see figure 3) is the result of interleaving planning with execution. It starts by generating plans according to available operations and time constraints. Besides the information from the business rules, for each atomic operation x we consider the estimated time for execution T_e^x and the probability of execution P^x. We also use a factor β^x that indicates if the corresponding task for operation x is optional ($\beta^x = 0$) or obligatory ($\beta^x = 1$) and which is initially provided by the domain ontology. Then, the optional tasks may be adjusted ($0 \leq \beta^x \leq 1$) by the current client in order to indicate how much the task is desired. In this manner, client preferences are captured.

Each plan has a quality factor which depends on the estimated execution time T_e^x for each operation, on the probability of success P^x, and on the preferences given by the client to the optional tasks[4]. When there are more than one generated plans, the best one according to this quality factor is chosen. The chosen plan may not be entirely followed, after each enactment of a decided service, a plan being chosen from the new generated plans.

In case a new plan is picked having the estimated time $T^{newplan}$ the extra time is uniformly distributed to each operation. The service corresponding to the first operation is queried and its answer is waited. If the answer does not assure the achievement of the task, the available time of the operation is decreased and the probability of success for that operation is updated. Even if $T^x_{available}$ remains positive, another plan may become the best one for the moment according to the F_p factor. In case the same plan is chosen again, there is no need to redistribute

[4] For instance, the optional operations like *updateHistory* may have $\beta = 0.5$.

begin
  currentplan = {}
  repeat
    $\mathcal{P}$=compose(*currentState*, *tasks*, Δ_{max})
    if $\mathcal{P} \neq \varnothing$ **then**
      for all $p \in \mathcal{P}$ **do**
        $F_p \leftarrow \sum_{k=1}^{N} \frac{\beta_x \cdot P^x}{T_e^x}$, $N =\mid p \mid$
      end for
      newplan $\leftarrow p$ with F_p maximal
      if *currentplan* $\neq$ *newplan* **then**
        $T_e^{newplan} \leftarrow \sum_{k=1}^{N} T_e^k$, $N =\mid newplan \mid$
        $T_s^{newplan} = \frac{\Delta_{max} - T_e^{newplan}}{N}$
        for all $x \in currentPlan$ **do**
          $T_{available}^x \leftarrow T_e^x + T_s^{newplan}$
        end for
      end if
      query the service associated to the first operation *op* of *newplan*
      answers $\leftarrow$ collectAnswers(op)
      if *updateState*(*currentState*, *answers*, *time*) $= \varnothing$ **then**
        $T_{available}^{op} \leftarrow T_{available}^{op} - time$
        $P^{op} \leftarrow update\_probability(T_{available}^{op})$
      else
        currentState=updateState(*currentState*, *answers*, *time*)
      end if
    end if
  until $\mathcal{P} = \varnothing$ or achieved(*currentState*, *tasks*)
end

Fig. 3. General algorithm for composition

the extra time. If no new plans can be generated and the requested *tasks* are not achieved, then the composition ends with failure.

4.2 Zeta agent

This agent has access to the web services descriptions and to the business rules, acting as an orchestrator for the composite service.

Planning phase. The process of generating the coordination artifact has two components that work together. A component that reasons above the domain task coordination rules and a lambda calculus component that works on states and tasks, trying to determine the sequence of actions related together by the states composition.

$$step : TASK \rightarrow (ComposedService \leftrightarrow ComposedService)$$

$$\begin{aligned}
&step = \lambda\, out : TASK \bullet \\
&\quad \{\Delta ComposedService \mid UniformOps \bullet \\
&\quad (\theta ComposedService, \theta ComposedService')\}
\end{aligned}$$

$$compose : \mathbb{P}\, ComposedService \rightarrow \mathrm{seq}\, TASK \rightarrow \mathbb{P}\, ComposedService$$

$$\begin{aligned}
&compose = \lambda\, init : \mathbb{P}\, ComposedService \bullet \lambda\, tasks : \mathrm{seq}\, TASK \bullet \\
&\quad \textbf{if } \#tasks = 0 \\
&\quad \textbf{then } init \\
&\quad \textbf{else} \\
&\quad \textbf{if } ((compose(init)(front(tasks))) \cap (\mathsf{dom}(step(last(tasks))))) \subseteq \varnothing \\
&\qquad \textbf{then } \varnothing \\
&\qquad \textbf{else } step(last(tasks))(\!|compose(init)(front(tasks))|\!)
\end{aligned}$$

The function *compose* generates the state corresponding to the composite service in the planning phase. It receives the sequence of required tasks and the time constraints and determines the sequence of transactions that solve these tasks. It is a backwards recursive process that generates on each step a relation of possible *before* and *after* states for all the known transitions that accomplish the current task, through the *step* function.

When a task can not be reached from any state transition, the Z agent tries to decompose a task in subtasks and corresponding actions. A simple decomposition could be done as in *DecomposeTask* action. This action is one of the *UniformOps* that it is tried on every step of the composition.

$$\begin{aligned}
&UniformOps == UCheckInsurance \lor UGetHistory \\
&\quad \lor UPayConsultation \lor UUpdateHistory \lor UDecomposeTask
\end{aligned}$$

DecomposeTask

$$\begin{aligned}
&\Delta ComposedService \\
&task? : TASK \\
&\hline
&\exists\, x : \mathrm{seq}\, TASK \mid x \mapsto task? \in rules \land \neg\, compose(\{\theta ComposedService\})(x) \subseteq \varnothing \bullet \\
&\quad \{\theta ComposedService'\} = compose(\{\theta ComposedService\})(x)
\end{aligned}$$

The relation generated by the *step* function is used to reach the next state of the *ComposedService*. The second condition of compose function expresses the situation when none of the *before* state for the task matches the current state of the ComposedService. In these cases, the empty set is retrieved, meaning the composition of that task is not possible. There are three cases where the composition may fail in the planning phase: *(i)* when there is no available transition for the task, meaning there are no service accomplishing the task, *(ii)* there are no decomposition rules, and *(iii)* from one intermediate state it is not possible to do the transition for the task.

UpdateStateCheckInsurance

$\Delta ComposedService$
$message? : \mathbb{N}$
$value? : BOOLEAN$
$clock? : \mathbb{N}$

$clock' = clock + time?$
$message? = checkInsuranceM$
$hasInsurance' = \textbf{if}\ value? = yes\ \textbf{then}\ yes\ \textbf{else}\ no$
$\quad \textbf{else}\ hasInsurance$
$payConsultation' = payConsultation;$
$op' = \varnothing$
$done' = done \cup (\textbf{if}\ value? = yes\ \textbf{then}\{checkInsurance\}\ \textbf{else}\{\})$

Fig. 4. Update state schema for checkInsurance Message

Updating phase. The response messages of the enacted services are received by the Z agent and the new state of the composite service is computed according to message translation rules. An *UpdateState* transition must be able to update the composite service state according to the outcomes of the services inquired at the current step.

Correctness of composition. All the states of the composed service are checked for validity according to the constraints expressed by domain task ontology and operation, respective message translation rules. The updating process of the service state following the receiving of a message is conditioned by the existence of a known transition possible in the current state that also follows the message translation rules. We check the acceptance of the transition given by the message by intersecting its *after* states set with all the possible *after* states from the current one through *stepEff* function. An unexpected message is considered to be valid for the composition process only if there is an available atomic transition taking the current state to the *after* update states.

$$updateState == \lambda now : \mathbb{P}\, ComposedService;\ message : \mathbb{N};\ value : BOOLEAN;$$
$$time : \mathbb{N};\ \bullet \{\Delta ComposedService \mid UniformUpdateState \bullet$$
$$(\theta ComposedService, \theta ComposedService')\}(\!|now|\!) \cap stepEff(now)$$

$$stepEff : \mathbb{P}\, ComposedService \rightarrow \mathbb{P}\, ComposedService$$

$$stepEff = \lambda beforeState : \mathbb{P}\, ComposedService \bullet$$
$$\{\Delta ComposedService \mid SimpleUniformOps \bullet$$
$$(\theta ComposedService, \theta ComposedService')\}(\!|beforeState|\!)$$

Interleaving planning with execution is the key element that assures the correctness of composition. If the service gets into an undesired state and compensation or rollback transitions are defined in the transition knowledge base, they will be included in the new plan, similar to the normal flow transitions.

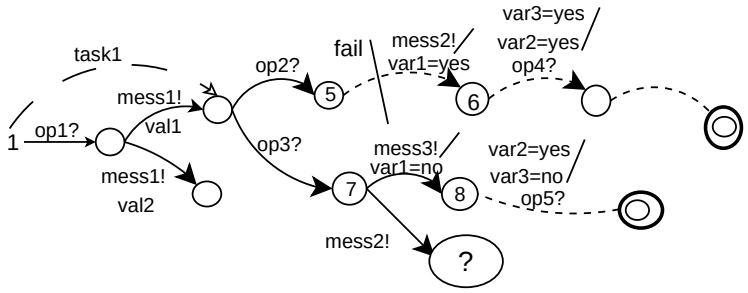

Fig. 5. A case of asynchronous message.

The composite service (figure 5) is represented as a sequence of states, transitions being determined by querying a service (*op1?)* or receiving an answer *(mess1!)*. A sequence of a transition querying a service and one receiving the corresponding message determines the achievement of a task (*task1*). The initial plan contains the request of *op2* that brought the composed service to the state 5 where *mess2* is the expected message. Due to the failure of receiving the expected message in the allocated time, a new plan is generated proposing *op3* as the first operation. The problem arises in state *7* where the composed service receives the expected message *mess3*, but also the message *mess2*. The wrong message is identified by the function *stepEff*, due to the fact that there is no defined atomic transition equivalent to one of those generated by the *updatingState* transition from state *7* and message *mess2*. The merging of states *6* and *8* is possible if their variables' values are consistent. In the case the merging is possible, there is no need to search back, it is enough to re-plan. In the contrary case, the state where the delayed message was expected must be identified and plans starting from both states *6* and *8* must be generated. The best one is followed.

The fulfillment of the requirements in each state constitutes another important component. The *achieved* function, receiving a service state and a task, checks if the specified *task* is accomplished in that state. A task is achieved if it is an atomic one, member of *done* variable, or there is a sequence of achieved subtasks that accomplishes the task. This function is used as a termination condition of the process.

$$
\begin{array}{|l}
achieved : \mathbb{P}\, ComposedService \times TASK \rightarrow \mathbb{P}\, BOOLEAN \\
\hline
achieved = \lambda\, currentState : \mathbb{P}\, ComposedService;\ task : TASK \bullet \\
\quad \{x : ComposedService \mid x \in currentState \bullet \\
\qquad \textbf{if}\ task \in x.done \vee \\
\qquad\quad (\exists\, seqx : \mathrm{seq}\, TASK \mid seqx \mapsto task \in rules \bullet \\
\qquad\quad \forall\, y : TASK \mid y \in \mathsf{ran}\, seqx \bullet yes \in achieved(currentState, y)) \\
\qquad \textbf{then}\, yes\, \textbf{else}\, no\}
\end{array}
$$

4.3 WSDL2Z agent

The goal of this agent is to support the dynamic integration of web services. It defines a middle-ware trading service for retrieving service instances that match a given service specification in Z language. To obtain the WSDL specifications available in the community, the WSDL2Z agent broadcasts requests of type *getWSDL*. The obtained WSDL specifications are translated into Z and they are compared with the model encapsulated in the proper domain agent. If the specifications match, the names of the operations available in the community and which are considered useful for the current composition are provided to the Zeta agent.

4.4 Domain agents

Domain agents provide a set of state variables and invariants for a specific domain. They have access to the domain task ontology and operation constraints rules. Identification of the domain task coordination rules from domain ontology is one of the responsibility of these agents.

For the medical domain, the *SimpleComposedService* type is defined by two domain variables *hasInsurance* and *payConsultation*, whilst the *InputData* type specifies four input variables of the composed service. Together with the types specific to the computation process *OpComposed* and *EffectComposed*, these two types define the ComposedService type (see section 3.3).

$$
\begin{array}{|l}
InputData \\
\hline
patient : QName \\
consultationFee : \mathbb{N} \\
diagnosis : \mathbb{N} \\
receipt : QName \\
\hline
\end{array}
\qquad
\begin{array}{|l}
SimpleComposedService \\
\hline
hasInsurance : BOOLEAN \\
payConsultation : BOOLEAN \\
\hline
\end{array}
$$

4.5 Reliability agent

The specification in Z of non-functional properties of the composite service are verified by this agent. The reliability of a web service represents the probability

that a request submitted to a service is correctly responded within the maximum expended time frame [9]. This agent has the task to compute the reliability value for each service from historical data about past invocations used when (i) the plans are generated, the zeta agent verifies or estimates if the plans can be executed within the time limit asked by the client and (ii) the OAA agent waits for answers only the time estimated by the reliability agent, after that it reports *NetworkFailure* and the zeta agent computes the next state.

4.6 Type-checking agent

Type checking is an important issue in web process composition. The process execution engines typically throw exceptions when they encounter incompatible types during data flow between activities. In order to use the output from one web service as input to another web service, it is often necessary to perform a data transformation. This agent acts as an intermediate layer between the client and the service. The agent converts the data type provided by the client and the data type supported by the service to Z representation and performs type checking. The agent ensures both that only compatible types are used while establishing a data flow, and also aids in decision making during the automated data flow.

5 Related Work

Automated composition of web services is an open research issue. On the one hand XML-based standards have been developed to formalize the composition of web services. This line is primarily syntactical and the interaction protocols are manually written. On the other hand, semantic approaches based on ontologies view the composition process as a goal oriented one, basically as a planning problem. Our approach starts from a formal specification is order to automate and validate the composition process, but also provides mechanisms to use ontological knowledge during the composition process. The main advantage relies on using the built-in state transition composition mechanism of the Z language. We advocate two strong points of the approach: i) it generates more reliable services, and ii) the composition expressivity in Z language is not limited to the reasoning capabilities of the description logic in OWL-S. Integration of agent technology with web services and semantic web is also aimed in [10] or [11]. In the former, the integration of web services is coordinated with TucSon, while in latter OWL-S descriptions of web services together with production rules are used by workflow managers, developed in JADE, that build or complete the workflow responding to the user requests. In our approach, the composition process is not limited to the reasoning capabilities of the description logic.

A classical approach for monitoring the service execution consists in planning as model checking [12]. Our zeta agent deals with such monitoring aspects, computing the current state using schema calculus. Specification of composite

services also uses service chart diagrams [13], providing a good number of control flow constructs. In our framework, the composition is verified at runtime, handling some unpredicted exceptions through re-planning.

Formal specification of complex systems uses a combination CSP-OZ-DC in [14] for the specification of processes, data and time. The WSAMI language was also proposed for the specification of a composite service [2]. Given the WSAMI specification of a service, an instance is automatically selected and composed upon a user request, according to the services that may be retrieved in the environment. In the web services context, the Z language was used for a formal specification of a constrained object model for the workflow composition [15]. The above research is focused on the design aspects, the resulted Z specification being compatible with UML. Our approach is more functional, the specification being animated by the Zeta tool and then executed by Z-based agents.

The value of flexible provisioning for service flows has been shown [16] by empirical evaluation in an experimental testbed.

6 Conclusions

In this paper, we introduce a framework for formal specification and verification of composite services. We considered failures associated with web services and we tried to handle such runtime exceptions by using formal methods. Using Z-based agents, a series of advantages exists: i) both process oriented knowledge and ontological knowledge are used in the composition process; ii) the operations of a service are represented by Z-schemes and the correctness of the composition is verified with the schema calculus; iii) the above mathematical framework is not limited to the reasoning capabilities of the description logic; iv) due to the existence of multi-agents, one can model more complex interactions between services, not only request-response messages;

We plan to enhance the animation capabilities of Zeta by introducing a tool for reasoning with the available Z specifications. Another challenge would be dealing with the preference concept for describing different importance levels for the composition rules. The framework is a step forward a functional system where the formal specifications and the semantic descriptions could work together for improving the collaboration between services in open environments.

Acknowledgments

We are grateful to the anonymous reviewers for useful comments. Part of this work was supported by the grant 27702-990 from the National Research Council of the Romanian Ministry for Education and Research.

References

1. Singh, M.P., Huhns, M.N.: Service-Oriented Computing:Semantics, Processes, Agents. John Wiley and Sons, Chichester West Sussex (2005)

2. Issarny, V., Sacchetti, D., Tartanoglu, F., Sailhan, F., Schibout, R., Levy, N., Talamona, A.: Developing ambient intelligent systems: A solution based on Web Services. Automated Software Engineering **12** (2005) 101–137
3. W3C: Web Services Description Language (WSDL) version 2.0 part 1: Core language. Technical report, W3C, available at http://dev.w3.org/cvsweb/ checkout /2002/ws/desc/wsdl20/ (21 February 2005)
4. Bultan, T., Fu, X., Hull, R., Su., J.: Conversation specification: A new approach to design and analysis of e-service composition. In: 12th International World Wide Web Conference (WWW'2003), Budapest, Hungary (2003) 403–410
5. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web services architecture. Technical report, W3C, available at http://www.w3.org/TR/2003/WD-ws-arch-20030808/ (8 August 2003)
6. Paurobally, S., Jennings, N.R.: Protocol engineering for web services conversations. Int J. Engineering Applications of Artificial Intelligence **18** (2005) 237–254
7. Jacky, J.: The way of Z - Practical Programming with Formal Methods. Cambridge University Press, Cambridge (1998)
8. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic composition of e-services that export their behavior. In: International Conference on Service Oriented Computing, Trento, Italy (2003)
9. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web service composition. In: 12th International World Wide Web Conference (WWW'2003), Budapest, Hungary (2003) 411–421
10. Morini, S., Ricci, A., Viroli, M.: Integrating a MAS coordination infrastructure with web services. In: Workshop on Web-Services and Agent-based Engineering at AAMAS, New York, NY, USA (2004)
11. Negri, A., Poggi, A., Tomaiuolo, M., Turci, P.: Agents for e-Business Applications. In: 5th International Joint Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan, ACM Press (2006) 907–914
12. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: ICAPS04, Workshop on Planning and Scheduling for web and grid Services, Whistler, Canada (2004)
13. Maamar, Z., Benatallah, B., Mansoor, W.: Service chart diagrams - description application. In: 12th International World Wide Web Conference (WWW'2003), Budapest, Hungary (2003)
14. Hoenicke, J., Olderog, E.R.: Combining specification techniques for processes, data and time. In Butler, M., Petre, L., Sere, K., eds.: Integrated Formal Methods. LNCS 2335. Springer-Verlag (2002) 245–266
15. Albert, P., Henocque, L., Kleiner, M.: A constrained object model for configuration based workflow composition. In: Business Process Management Workshops. (2005) 102–115
16. Stein, S., Jennings, N.R., Payne, T.R.: Flexible provisioning of service workflows. In: 17th European Conference on Artificial Intelligence. (2006)