Formulation of Hierarchical Task Network Service (De)composition

Seiji Koide[1] and Hideaki Takeda[2]

[1] The Graduate University for Advanced Studies (SOKENDAI), 2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan
koide@nii.ac.jp,
WWW home page: http://www-kasm.nii.ac.jp/~koide/
[2] National Institute of Informatics and SOKENDAI, 2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan
takeda@nii.ac.jp

Abstract. The Hierarchical Task Network (HTN) planning method is conceived of as a useful method for Web service composition as well as classical task planning. However, there are no complete successes of service composition by HTN as yet. The reason is the Web service composition process involves the interactive dataflow between variables in preconditions and input/output parameters of services. Since values of variables in service IOPEs are required to compose services with the classical HTN method, we are compelled to perform services in the composition process. However, we cannot perform any world-altering Web service in practice in the composition process. In this paper, we address more radical approach of HTN planning method for Web service composition and decomposition with premising the openness and uncertainty of WWW. We capture composite services, which include abstract meanings with respect to the variables of Web services, as abstract programs to be tailored to individual users and instantiated to executable programs. In this view, the Web service composition process can be conceived of as a sort of automated programming process, and HTN is deemed as a structured workflow or a prototype of actual programs. In this paper, we formalize Web service composition/decomposition by HTN method using the term of satisfiability of situation calculus, and address the algorithm for Web service (de)composition that does not require performing services.

1 Introduction

The Hierarchical Task Network (HTN) planning method is conceived of as a useful method for Web service composition as well as task planning, and several works on the web service composition have been attempted with HTN [6]. However, there are a few but serious discrepancies between task planning and service composition. First, a web service involves inputs and outputs in addition to the precondition and the effects. Second, the constraint in web service composition is not partial-orders of tasks but control constructs, in which a subtask may be

composite and there are dataflows and control flows between subtasks. Therefore, the semantics of Web service composition are much more complex and analogous to automated programming rather than automated planning. The realization of the service composition ought to differ from that of task planning.

Sirin, et al. [11] achieved the Web service composition using the HTN planning method. They invented the translator from the OWL-S service description to the SHOP2 [6] task planning domain. In order to enable the translation from the web service composition domain to the HTN task planning domain, they assumed that an atomic Web service is either a strict information-providing Web service, which does not have the effects, or a world-altering Web service, which does not have outputs, but they do not assume world-altering information-providing services.

This assumption is ascribed to the fact that they used SHOP2, which was originally developed for classical task planning problems, and they did not reformalize HTN method for Web services. A precondition of method/operator in HTN is evaluated to test whether the precondition hold on the state or not. In case that a variable in a precondition is unified to some output datum of a Web service in planning, we need to know the value of the output. They state that we do not want to actually affect the world during planning, and do want to gather information from information-providing Web Services.

Besides the complexity in the service composition, Semantic Webs stand on the assumption that the world is open and dynamic. Therefore, we must consider the uncertainty of the world in service composition and execution processes. More precisely, we cannot expect service preconditions that hold in a composition process also hold in a execution process. As a result, we cannot help but abandon the soundness and completeness of planning when we consider both composition and execution processes in the dynamic world. In fact, the authors embrace the problem how to deal with the progression of situation in our application [5] from light anomalies in planning time to heavy anomalies in execution time and from one control mode to another control mode in the rocket launch operation process.

The authors claim that the uncertainty of WWW must be coped with by Web service agents situated in circumstances instead of planning systems. Such an agent is known as situated planning agent [9], which can replan and adapt failed plans to changeable situation. The behavior under the uncertainty is the common observation among animals and intelligent human beings in the real world, and it is the same on even Web service agents in use.

Under the premise of the situated planning agent in the future, in this paper, we formalize the web service composition and decomposition with expanding the HTN formalization using the idea of automated planning [2] and situation calculus [8]. In Section 2, we review the formalization of task planning by HTN. Then, we expand it for Web service composition. We define the concept of applicability, satisfiability, executability for Web services from the viewpoint of situation calculus, and address an algorithm for service composition/decomposition. In Section 3, we describe the implementation of the service (de)composition algo-

rithm, which is straightforwardly embodied of the algorithm using undeterministic search technique with continuation. The algorithm is incomplete because of the open world assumption but do not compel to perform services in planning. In Section 4, we discuss the related work, and we conclude at Section 5.

2 Formalisation of HTN for Web Services

2.1 Hierarchical Task Network Planning

In this subsection, we review the classical task planning by HTN according to the description in [2]. The expansion of HTN to web service composition and decomposition is described after the next subsection.

Let $\mathcal{L}$ be a first-order language for planning, in which there are predicate symbols, constant symbols, and variable symbols. If an atomic formula, which does not contain logic connectives, does not contain any variable symbol, it is called *ground atom*, otherwise *unground atom*.

A *state* s is a set of ground literals, i.e., ground atoms or negations of ground atoms in $\mathcal{L}$. S denotes a set of states. An atom p holds in s iff $p \in s$.

Definition 1. *A planning operator in task planning is a triple such that*

$$o = \langle name(o), precond(o), effects(o) \rangle.$$

- $name(o)$ is a name of operator, which has a syntactic expression of the form $n(x_1, ..., x_k)$ where n is a unique symbol called *operator symbol*, and $x_1, ..., x_k$ are all of variable symbols that appear anywhere in o.
- $precond(o)$ is the preconditions of o, and $effects(o)$ is the effects of o. Both are a set of literals, i.e., atoms and negations of atoms.

Definition 2. *An HTN method in task planning is a 4-tuple, that is,*

$$m = \langle name(m), task(m), subtasks(m), constr(m) \rangle.$$

Where $name(m)$ is an expression of the form $n(x_1, ..., x_n)$, n is a unique symbol for the method, and $x_1, ..., x_n$ are all of variables that occur anywhere in m. A set of pair $\langle subtasks(m), constr(m) \rangle$ makes a task network.

A task is an expression of the form $t(r_1, ..., r_k)$ like the name of operator but has a different name from and fewer parameters than the name of method. t is called *task symbol*, and $r_1, ..., r_k$ are terms. Every operator symbol is a task symbol, and every operator can be a task. When a task symbol t is an operator symbol and its terms can be unified to variables of the operator, the task is called *primitive*, otherwise it is *nonprimitive*. A task in method is nonprimitive.

Definition 3. *A task network in task planning is a pair*

$$w = \langle U, C \rangle.$$

Where U is a set of all of task nodes in the network and C is a set of constraint such as partial task ordering and preconditions for tasks.

Each task node $u \in U$ contains a task t_u. If all of tasks in U $\{t_u \mid u \in U\}$ are ground, then w is called ground, otherwise w is unground. If all of tasks $\{t_u \mid u \in U\}$ are primitive, then w is called primitive, otherwise nonprimitive.

2.2 Web Service Network by HTN

In this subsection and hereafter, we formalize Web service network by extending HTN task planning described above.

We expand the definition of *state in Web services* so that it includes not only atoms from precondition and effects but also web service inputs and outputs. Outputs of Web services are added into the state as well as positive effects. Note that inputs of Web services are taken from atoms in the state and/or outputs of other services. The data stream from an input to an output via services is called *dataflow*, and we call a datum that streams in and out via services *IO fluent*.

Definition 4. *An atomic service is an expansion of the operator, and defined as a 5-tuple such that*

$$as = \langle name(as), inputs(as), outputs(as), precond(as), effects(as) \rangle.$$

- $name(as)$ is a name of service. As well as operator name $name(o)$, it has a syntactic expression of the form $n(x_1, ..., x_k)$, but variable symbol x_i comes from not only $precondition(as)$ and $effects(as)$ but also $inputs(as)$ and $outputs(as)$.
- $inputs(as)$ denotes inputs to the web service as, and $outputs(as)$ denotes outputs returned by the web service. $precond(as)$ represents preserving or causal condition of as for the web service execution. $effects(as)$ is the side effects or causal effects onto the state s by the web service execution. $inputs(as)$ and $outputs(as)$ is an sequence of variables respectively, while $precond(as)$ and $effects(as)$ is a set of literals, as same as in operators.

Definition 5. *A composite service is an expansion of the method in HTN task network, and defined as a 7-tuple, that is,*

$$cs = \langle name(cs), task(cs), inputs(cs), outputs(cs), subtasks(cs), \\ precond(cs), controlConstruct(cs) \rangle.$$

Where the definition of $name(cs)$, $inputs(cs)$, $outputs(cs)$, and $precond(cs)$ are same as the definition of that in atomic service, but the notation of $task(cs)$, $subtasks(cs)$, and $controlConstruct(cs)$ firstly appear here in a composite service.

A task in Web services is defined as same way in HTN task planning. If the task symbol t is a name symbol of $name(as)$ of atomic service as and its terms $r_1, ..., r_n$ can be identical to variables of the atomic service, the task is called primitive.

A task network in Web service is a pair of a set of $subtasks(cs)$, and a set of $controlConstruct(cs)$.

Definition 6. *A task network in Web service composition and decomposition is expressed as*

$$w = \langle U, CC \rangle.$$

Where U is a set of all of task nodes in the network, and CC is a set of all control construct included in the network. Note that the constraint of CC includes control flows, dataflows, and preconditions of task-corresponding composite services and atomic services. The $controlConstruct(cs)$ contains task flows (abstract control flows) and dataflows. A composite service can contain only one control construct in definition, but a control construct can contain other tasks and control constructs in the specific form of various kind of control constructs, such as *sequence*, $ifThenElse$, $loopWhile$, etc.

We can consider various $controlConstruct$, but in this paper we define only three as follows.

Definition 7. *A sequence is a tuple of any number of subtasks.*

$$seq = \langle elt_1(seq), elt_2(seq), ..., elt_k(seq) \rangle$$

Where $elt_i(seq), i \leq k$ is a subtask in the cs. In the execution of *sequence*, $elt_i(seq)$ is performed in the order of the sequence.

In HTN task planning, constraint $constr(m)$ represents partially orders of subtasks. Therefore, a predecessor and the successor of tasks can be interleaved by another task. However, no service can part the task sequence in sequential performance of Web services.

Definition 8. *An $ifThenElse$ is a 3-tuple as follows.*

$$ite = \langle if(ite), then(ite), else(ite) \rangle$$

Where $if(ite)$ is a condition that does not cause any side effect in evaluation, and $then(ite)$ is a subtask or a control construct that is performed when $if(ite)$ is true in the state. Optional $else(ite)$ is a subtask or a control construct that is performed when $if(ite)$ is not true. This $ifThenElse$ has two possibility of branching of control in the execution process.

Note that $then(ite)$ is performed if and only if the condition of $if(ite)$ is true in the state, but $else(ite)$ is performed not only in case of negation of the condition of $if(ite)$ but also in the case of unknown condition for $if(ite)$, with the premise of the open world assumption of Web Semantics.

Definition 9. *$loopWhile$ is a 2-tuple such as*

$$lw = \langle while(lw), seq(lw) \rangle$$

Where $while(lw)$ is a condition during which is true $seq(lw)$ is performed repeatedly. $seq(lw)$ is a *sequence*. Note that performance of *sequence* is terminated even if $while(ls)$ becomes unknown in the execution process.

2.3 Web Service Composition and Decomposition by HTN

On one hand, the objective of task planning is to obtain totally ordered sequences of actions that achieve a goal, where a goal g in task planning is a set of ground literals. On the other hand, the objective of Web service composition and

decomposition is to obtain a set of executable task flows or programs that are an instantiated network of procedure in the environment. We have two categories of goals in essence in Web service composition, i.e., the alteration of world by Web services and the information retrieval that is provided by performance of Web services. The former is the same as classical task planning but the latter is different from goals in classical task planning. We give ground literals (say, instances of classes in OWL) as goals in task planning, but we cannot designate values of service outputs as goals in Web service composition. The values of output variables are the very thing we want to know. We are able to only designate typed variables (classes in OWL) as goal. Web service composers must generate executable instantiated task flow that includes atomic (invokable) services that achieve world-altering goals and information-retrieval goals. Furthermore, the coupling of world-altering service performance and information-retrieval service performance may happen.

In this subsection, we discuss the intepretability and composability of services from the standpoint of satisfiability in situation calculus [8].

Satisfiability of Web Service: In task planning, an operator o is an abstract one that stands for all instance operators named by an operator symbol n. In the instance operators of o, variable symbols in $name(o)$, $precond(o)$, and $effects(o)$ are substituted with corresponding constant symbols. If an instance operator contains ground atoms and does not contain unground atoms, it is called *ground*, otherwise *unground*. A ground operator that includes ground atoms in S is called *action*.

To discuss the interpretation of assertions for Web service (de)composition, we set an state transition machine. Let Σ be an state transition machine [2]. We consider a mapping from a state s in assertions of the planning problem P to a state in Σ. For a state s described in planning language $\mathcal{L}$, the corresponding state in Σ is denoted by $s^{\mathcal{I}}$, and $\mathcal{I}$ is called *interpretation*. In case that there are a mapping from s_{i-1} to $s_{i-1}^{\mathcal{I}}$ and s_i to $s_i^{\mathcal{I}}$, if there is a mapping from an instance operator o in P that links from s_{i-1} to s_i to $o^{\mathcal{I}}$ that links from $s_{i-1}^{\mathcal{I}}$ to $s_i^{\mathcal{I}}$, it is called *satisfiable* on the operator o.

We expand this interpretation to Web service composition and decomposition. In case that there is a mapping from s_{i-1} to $s_{i-1}^{\mathcal{I}}$ and s_i to $s_i^{\mathcal{I}}$, and from an atomic web service as in P that links from s_{i-1} to s_i to $as^{\mathcal{I}}$ that links from $s_{i-1}^{\mathcal{I}}$ to $s_i^{\mathcal{I}}$, it is called *satisfiable* on the atomic web service as.

For an atomic service as, if and only if the precondition $precond(as, s)$ is satisfiable, namely a condition in $precond(as)$ holds in an interpretation $s_i^{\mathcal{I}}$, and inputs $inputs(as, s)$ is satisfiable, namely we can find the unification for each variable of inputs onto $s_i^{\mathcal{I}}$, where the unification is identical to that for the $precond(as)$, then the atomic service as is satisfiable on s.

$$s \models as \Leftrightarrow (s \models precond(as) \ \wedge s \models inputs(as))$$

On the other hand, a composite service cs is satisfiable, iff $precond(cs)$, $inputs(cs)$, and $controlConstruct(cs)$ is satisfiable.

$$s \models cs \Leftrightarrow (s \models precond(cs) \ \wedge s \models inputs(cs) \ \wedge s \models controlConstruct(cs))$$

When the precondition and inputs of a service are satisfiable, let us call the service *applicable*. An applicable atomic service is always satisfiable but an applicable composite services are not necessarily satisfiable. In other words, an applicable atomic service has a model on s but there is a case that an applicable composite service has no model on s.

In order to know the satisifiability of a composite service, we need to know the satisfiability of *controlConstruct*.

Satisfiability of Task: If a task $t(r_1, ..., r_n)$ is primitive, then the task $t(r_1, ..., r_n)$ is applicable and satisfiable, iff as is satisfiable.

$$s \models t \Leftrightarrow ((name(as) = \sigma(t(r_1, ..., r_n), \theta)) \wedge (s \models as))$$

Where $\sigma(t(r_1, ..., r_n), \theta)$ expresses the substitution of $t(r_1, ..., r_n)$ by unifier θ for s.

If a task is nonprimitive and the corresponding service is composite, then the task $t(r_1, ..., r_n)$ is satisfiable, iff cs is satisfiable.

$$s \models t \Leftrightarrow ((name(cs) = \sigma(t(r_1, ..., r_n), \theta)) \wedge (s \models cs))$$

Satisfiability of *sequence* Control Construct: Let be seq a *sequence*, and let s_0 the initial state for *sequence* seq. Let be $\gamma(s_{i-1}, elt_i(seq))$ a progression of subtask $elt_i(seq)$ for state s_{i-1}. It is seemed that the execution of task $elt_i(seq)$ for state s_{i-1} makes a progression and yields the next state s_i. However, we cannot really execute the task in service composition processes. Instead we make the progression by the unification that change abstract types of variables to more special types.

$$s_i = \gamma(s_{i-1}, elt_i(seq))$$

This is the same meanings of the following expression on action by Reiter [8], whereas there are only actions, that contain preconditions and effects, and no services that contain inputs and outputs.

$$s_i = do(elt_i(seq), s_{i-1})$$

Let us express the execution of $elt_i(seq)$ by inputs $in^1_{i-1}, ..., in^k_{i-1}$ as $elt_i(seq)(in^1_{i-1}, ..., in^k_{i-1})$. If $elt_1(seq)$ is satisfiable for s_0 and inputs $in^1_0, ..., in^k_0$, then we can make a progress for s_0 and obtain s_1 for $elt_1(seq)(in^1_0, ..., in^k_0)$ by making a progress of unification instead of performance of $elt_1(seq)$.

$$s_1 = \gamma(s_0, elt_1(seq)(in^1_0, ..., in^k_0))$$

Then, if $elt_2(seq)$ is satisfiable for s_1 and inputs $in^1_1, ..., in^l_1$, we can obtain the next state s_2.

$$s_2 = \gamma(s_1, elt_2(seq)(in_1^1, ..., in_1^l))$$

We cannot say that if all tasks $elt_i(seq)(in_{i-1}^1, ..., in_{i-1}^k)$ in *sequence* are independently satisfiable for each states then the *sequence* is satisfiable, because the satisfiability of the control construct also depends on the dataflow. We represent this constraint of dataflow in control construct $dataflow(seq)$. If the dataflow constraint is held correctly in the progression of control constructs, we call the control constructs has a model. Thus,

$$s_0 \models sequence \Leftrightarrow (s_0 \models dataflow(seq) \ \wedge \{s_{i-1} \models elt_i(seq) \mid i = 1, ..., k\})$$

Note that each task $elt_i(seq)$ may be composite and its satisfiability is decided with the satisfiability of the corresponding composite service. Obviously, this definition for the control construct and the composite service is recursive but not tautology, because the satisfiability can be decidable, namely the composite service is decomposed down to atomic services in an acyclic task network, and every atomic service is decidable for satisfiability.

Satisfiability of $ifThenElse$ Control Construct: Let be ite an instance of $ifThenElse$, and s_{i-1} is the state for ite. Then, we have three possibilities on the satisfiability of ite.

- One is for positive condition such as
$$s \models ite \Leftarrow (s \models if(ite) \ \wedge s \models then(ite)).$$
- Another is for negative condition such as
$$s \models ite \Leftarrow (s \models \neg\, if(ite) \ \wedge s \models else(ite)).$$
- The last is for unknown condition such as
$$s \not\models ite \Leftarrow (s \not\models if(ite) \ \wedge s \not\models \neg\, if(ite)).$$

If $if(ite)$ holds and $then(ite)$ is satisfiable, then the ite is satisfiable. If the negation of $if(ite)$ holds and $else(ite)$ is satisfiable, then ite is satisfiable. However, when the condition of $if(ite)$ is unknown, we cannot deduce that ite is satisfiable. In the execution process, the value of $if(ite)$ is known as a result of the service execution and progression of states, but it may be unknown in composition processes. Therefore, the composer cannot proceed the reasoning at this branch of possibility. In such a case, the agent has a selection according to two strategy, i.e., speculative strategy and assurance strategy. In the speculative strategy, the agent takes one of the possibilities of branches. The instantiated program as a result of (de)composition may be not valid in execution. If the executer failed the execution of the generated program, the agent repairs the failure and replans at the point. In the assurance strategy, the agent brings the incomplete programs to an execution phase and executes it. The agent restart planning when the value of the $if(ite)$ is known. Thus, the functionality of incremental planning and replanning is requisite for the agent in the open world. Generally speaking, it is useful to positively give rich information of negation, disjunction, and complementness in ontologies for the open world.

Satisfiability of *loopWhile* Control Construct: Let be s_0 an initial state for an instance of $loopWhile$, lw. When while condition $while(lw)$ of lw does not hold via knowing the negation of $while(lw)$, lw is satisfiable and the state s_0 does not evolve. If while condition $while(lw)$ holds for s_0, then lw is satisfiable iff $seq(lw)$ is satisfiable. However, when it is unknown whether $while(lw)$ hold or not, we treat it in the same way as $ifThenElse$.

As a result of the execution of satisfiable sequence $seq(lw)$, the state evolves and this process is repeated again while $while(lw)$ is satisfiable for the evolving state in the loop.

$$
\begin{aligned}
s \models lw \Leftarrow\ & s \models \neg\ while(lw) \\
s \models lw \Leftarrow\ & s \models while(lw)\ \wedge seq(lw) \\
s \not\models lw \Leftarrow\ & s \not\models while(lw) \wedge \not\models \neg while(lw)
\end{aligned}
$$

Note that even if the same procedure is executed repeatedly, the states evolve. Similarly, even if the same procedure is tested repeatedly on the satisfiability, the states may evolve because types of variables evolve more precisely step by step by loop using the typed unification, which is described later.

2.4 Algorithm of Web Service Composition and Decomposition

We cannot decompose the control construct of composite services into subtasks as HTN task planning does. Instead, we collect all of satisfiable bindings of variables that make a control construct in hand satisfiable, and search all possibilities by substituting all variables in service parameters. Here note that inputs and outputs are typed in DL or OWL, and variables and constants in precondition and effects also typed. Therefore we need typed unification to compute satisfiability. The typed unification algorithm is described in the next section.

Suppose that u is a task node in U $\{u \in U\}$, cs is a composite service, and $task(cs) = t_u$. Then cs instantiate u as producing the instantiated task network w'. If $w = \langle U, CC \rangle$ is primitive, and CC is satisfiable for s, then w is a solution for s such that the executer can execute CC for s. If $w = \langle U, CC \rangle$ is nonprimitive (i.e., at least one task in U is nonprimitive), then w is a solution for s if there is a sequence of satisfiability checking that simulates dataflows and control flows in control constructs, and a primitive task network w' is obtained as a result.

The algorithm of HTN Web service composition and decomposition is described as follows.

procedure SWHTN(s, w', w, AS, CS, D)
  if $w = \emptyset$ **then return** w'
  nondeterministically choose any $u \in U$ that has no predecessors in w
  if t_u is primitive **then**
    $active \leftarrow \{\sigma(s) \mid as = \text{discover}(t_u, w, AS, D)$ and as is satisfiable in s
      σ is a substitution with satisfiable bindings of as
    if $active = \emptyset$ **then return** fail()
    nondeterministically choose any $\sigma(s) \in active$

return SWHTN($\sigma(s), w' + \sigma(u), \sigma(w - \{u\}), AS, CS, D$)
else ;; t_u *is nonprimitive.*
$active \leftarrow \{\sigma(s) \mid cs =$ discover(t_u, w, CS, D) and cs is applicable in s
σ is a substitution with applicable bindings of cs
if $active = \emptyset$ **then** return fail()
nondeterministically choose any $\sigma(s) \in active$
if t_u is not satisfiable for $\sigma(s)$ **then return** fail()
return SWHTN($\sigma(s), w' + \sigma(u), \sigma(w - \{u\}), AS, CS, D$)

Where s is a state in a situation, w is a part of task network that is to be instantiated. w' is a part of task network that is instantiated. AS is a set of atomic services, and CS is a set of composite services. D is a domain knowledge of the target field. This algorithm contains the loop of SWHTN() via tail recursive call. $fail()$ causes automatic backtracking to the point of the last choice of nondeterministic selection.

The algorithm is very similar to HTN of task planning [2] and SHOP2 [6], because we already have a convenient terminology *satisfiable*. We used it instead of the terminology of *ground action* in HTN task planning in order to collect *active* atomic service. However, there is no decomposition of composite tasks. Instead we substitute parameters with bindings in the situation.

In the worse case, we cannot obtain complete solutions from this algorithm, because it is possible that we encounter unknown conditions without the execution of Web services. In such a case, the agent resolves the problem in the manner of the situated planning agent [9].

3 Implementation

3.1 State Space and Variables

Generally, an atom can be expressed as form $t(r_1, ..., r_{n-1}, r_n)$. Then, if t is functional such that the combination of predicate t and variables $r_1, ..., r_{n-1}$ has a mapping to each different r_n on each state s, t is called *fluent*. r_n is called *state variable* and expressed as $r_n = t(r_1, ..., r_{n-1})$. A fluent can be also expressed as $t(r_1, ..., r_{n-1}, r_n) = true$ or simply $t(r_1, ..., r_{n-1}, r_n)$. A strong negation of atom can be expressed as $t(r_1, ..., r_{n-1}, r_n) = false$. On the close world assumption, the absence of positive atom means the negation of the assertion. On the open world assumption, it means unknown on the assertion.

The state is expressed as a list of state variables as atom. The following shows an example of the state in which an individual `Lucy` has an appointment, and `HAL` has also an appointment in Lisp. Note that `Lucy` is already defined as individual of `Person`, and `HAL` is already defined as individual of `Robot`.

```
(setq *state*
      (make-state '((hasAppointment(Lucy) = LucysAppt)
                    (hasAppointment(HAL) = HALsAppt))))
```

On the other hand, we can make a typed variable in the following form in our system.

```
(make-condition '((hasAppointment((?p - Person)) = ?appt)))
```

Note that this appointment with variables typed `Person` is unified with `Lucy`'s appointment but not unified with `HAL` appointment.

3.2 Typed Unification

The unification algorithm by Russel and Norvig [9] is expanded to accept OWL entities, classes and individuals. A variable is also an individual of a class in the domain. Consider the followings, where variablep(x) tests whether x is a variable individual and variable?(x) tests x for the lisp symbol of individual.

function Typed-Unify(x, y, θ)
  if θ = failure **then return** failure
  else if $x = y$ in semantics of OWL **then return** θ
  else if variablep(x) **then return** Typed-Unify-Var+(x, y, θ)
  else if variablep(y) **then return** Typed-Unify-Var+(y, x, θ)
  else if variable?(x) **then return** Typed-Unify-Var(x, y, θ)
  else if variable?(y) **then return** Typed-Unify-Var(y, x, θ)
  else if compound?(x) and compound?(y) **then**
    return Typed-Unify(Args(x), Args(y), Typed-Unify(Op(x), Op(y), θ))
  else if list?(x) and list?(y) **then**
    return Typed-Unify(Rest(x), Rest(y), Typed-Unify(First(x), First(y), θ))
  else return failure

As shown below, the algorithm of Typed-Unify-Var looks like the same as original Unify-Var in [9] at the surface level, but making a new binding $\{var/x\}$ differs from the original. In addition to the symbol level binding between var and x, the class level bindings are taken into account. If two classes of var and x are disjoint each other, then no unification is made and failure is returned. If two classes relates each other in subsumption relation, then a mapping to the specific class is made. Otherwise, a mapping to the intersection of both classes is made.

function Typed-Unify-Var(var, x, θ)
  if $\{var/val\} \in \theta$
    then return Typed-Unify(val, x, θ)
  else if $\{x/val\} \in \theta$
    then return Typed-Unify(var, val, θ)
  else if var occurs anywhere in x
    then return failure
  else return make $\{var/x\} \in \theta$

Typed-Unify-Var+ is prepared for the binding of OWL individual objects. In the integration to SWCLOS [4], an variable var in Typed-Unify-Var+ is an object typed to OWL classes in the domain, while x may be an individual of domain classes or may be an variable typed to a domain class.

function Typed-Unify-Var+(var, x, θ)
  if x is individual **then**
    if disjoint?(class(var),class(x)) **then return** failure
    else if class(var) = class(x) in semantics of OWL
      then return make $\{symbol(var)/symbol(x)\} \in \theta$
    else if subsumed?(class(x),class(var))
      then return make $\{symbol(var)/symbol(x)\} \in \theta$
    else if subsumed?(class(var),class(x))
      then return make $\{symbol(x)/$individual(class(var)) $\} \in \theta$
                  make $\{symbol(var)/symbol(x)\} \in \theta$
    else return
      then return make $\{symbol(x)/$individual(intersection(class(var),class(x)))$\} \in \theta$
                  make $\{symbol(var)/$individual(intersection(class(var),class(x)))$\} \in \theta$
                  make $\{symbol(var)/symbol(x)\} \in \theta$
  else return failure

Through unification, the type of variable is specified step by step. The value of variables are bound to abstract concepts to special concepts along with the progression via unification. However, if we have poor ontologies with respect to the class hierarchy, this unification easily leads silly results. For example, if there is no knowledge that xsd:integer is disjoint with xsd:float, the intersection of xsd:integer and xsd:float is resulted. However, if there is an assertion that ship is disjoint with automobile, the system fails to find the route by amphibious-vehicle.

3.3 Nondeterministic Choice by Continuation

The continuation is well known as program control technique in Scheme. In short, it is a program frozen in action [3]. When the computational object that contains the state of a frozen computation is evaluated, it is restarted where it left off. This machinery can be a great help to implement the exception handler, multiprocessing, and nondeterministic search and choice. In order to integrate the (de)composer with SWCLOS [4], which is an OWL Full reasoner and language built on top of Common Lisp Object System, we have adopted the technique of continuation in Lisp [3]. The (de)composition algorithm SWHTN in Subsection 3.4 can be straightforwardly implemented with the continuation.

4 Related Work and Concluding Remarks

4.1 Reasoning from Action to Service

Reiter [8] enlightened the task planning problem from situation calculus. Berardi et al. [1] discussed the synthesis of Web services from situation calculus, but the work still stays in the closed world assumption. Sohrabi et al. [13] demonstrated Web service composition using agent programming language GoLog, which is based on situation calculus. However, the problem of the interaction between precondition and inputs/outputs, which is posed by Sirin et al. in service composition using SHOP2 [11], seems to be still open.

All of them including [11] strongly stick the validity and completeness of service composition. However, the authors argue that the openness and uncertainty of WWW lead us to the incompleteness of composition. The problem must be solved by the intelligent behavior of agent in the changeable world. In this paper, we formalized Web service composition/decomposition by HTN using the idea of satisfiability in situation calculus, and addressed the algorithm for service composition with the directive suggestion of implementation. We also suggested that we need situated planning agent that adaptively behaves in service composition, decomposition, execution, and monitoring, by introducing the open world assumption into the formulation.

Sirin and Parsia [12] deeply discussed the integration of OWL and the task planner. In a sense, it could be said that this paper is a legitimate argument on the HTN formalization touched by them. We addressed the typed unification to make a progress about variable bindings. The system is based on SWCLOS [4] for OWL reasoner. Sirin and Parsia pointed out the existentially bound variables in preconditions may cause the disparity of binding between planning time and execution time. We have no solution on this problem in this paper. We know that SWCLOS cannot reason out on the existential quantifier. On the other hand, the problem on the creation of anonymous individuals is easily solved with SWCLOS, because SWCLOS is built on top of Common Lisp Object System (CLOS) and every concept and individual is an object, even if it is anonymous or not.

4.2 Framework of Web Service Agent

In this paper, we often mentioned the composer and executer. Fig. 1 depicts a framework of a web service agent by HTN service composition. The system consists of five modules, a composer, a decomposer, an executer, the memory, and the user interface.

The composer composes an appropriate task flow on user demands, using the library of workflows or business patterns. The decomposer instantiates the abstract task flow to executable programs that include the invocations of Web services. The executer interprets and executes the instantiated programs with invoking Web Services. The machinery functions as memory for various internal data of agent, i.e., plans of tasks, simulating data of states under planning,

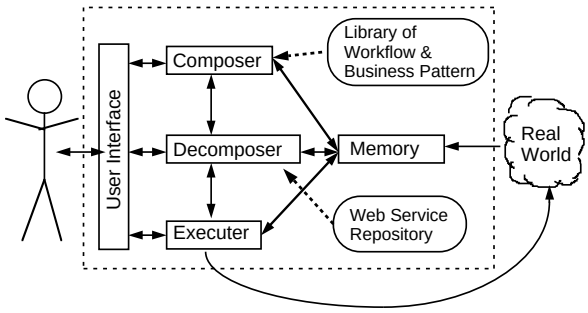

Fig. 1. Agent Framework

results of execution, etc. Some part in memory reflects the variations of outer world with sensing data and poling queries, etc.

The user interface works for the communication between the agent and a user. Some ambiguity and nondeterministic choice in task planning may be solved with the help from the user through this interface.

This agent plays double roles as intelligent agent for invoking Web services and interface agent for users. In fact, we have almost developed such an agent in our application of Rocket Launch Operation Support System [5], in which the interface agents that mediate between support engineers in distance from the launch site and various Web services for information providing on rocket launch process help the engineers to decide the counteraction in the contingency of rocket launch process. Semantic Web service technology is adopted to realize the flexible intelligent decision support system.

4.3 Web Service Discovery

Web service subsumption hierarchy (not task-subtask hierarchy) can be utilized to discover appropriate Web services that are annotated with inputs, outputs, precondition, and effects. If a Web service subsumption hierarchy is richly and consistently built with domain ontologies, the discovery of Web services is just same as the calculation process of subsumption and classification in DL or OWL on an unknown service. If inputs and outputs are annotated just as xsd:integer or xsd:string like Java program language, it produces no effects on service discovery. If IOPEs are typed to classes in middle level ontologies and enter prize ontologies, the agent can understand meanings of services. If the Web service annotation is poor or inconsistent with domain ontologies, agent must nondeterministically solve this problem. Sycara, et al. [10], discussed this problem and presented OWL-S Matchmaker and OWL-S Virtual Machine.

In this paper, we formulated HTN Web service composition and decomposition on the assumption that agent can automatically and appropriately discover

a number of Web services in the situation. The function discover in the algorithm shown above is such a thing.

4.4 Web Service Decomposition

Usually, HTN is conceived as a method for Web service composition. However, the composition process in HTN is strictly coupled with the decomposition process. We consider the agent depicted in Fig. 1, in which the composer composes Web services from scratch. In this service composition, the partial ordered planner technique like UCPOP [7] will be useful rather than HTN. In this paper, we concentrated the discussion to HTN task planning process, in which we have a top task of HTN at first, and the top task and related abstract tasks are instantiated by selecting and combining subtasks and reducing the ambiguity of task parameters step by step. We call this HTN planning process *service decomposition*.

References

1. Berardi, D., Calvanese,D., Giacomo, D., Mecella, M.: Reasoning about Actions for e-Service Composition. International Conference on Automated Planning & Scheduling, ICAPS 2003 (2003)
2. Ghallab, M., Nau, D., Traverso P.: Automated Planning Theory and Practice. Morgan Kaufmann (2004)
3. Graham, P.: On Lisp. Prentice Hall, (1993)
4. Koide, S. Takeda, H.: OWL-Full Reasoning from an Object Oriented Perspective. Asian Semantic Web Conf., ASWC2006 (2006) 263–277
5. Misono, S., S. Koide, N. Shimada, M. Kawamura, and S. Nagano: Distributed Collaborative Decision Support System for Rocket Launch Operation. IEEE/ASME Int. Conf. Advanced Intelligent Mechatronics, AIM2005, (2005)
6. Nau, D., T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman: SHOP2: An HTN Planning System. J. Artificial Intelligence Research, **20**-12, (2003) 379–404
7. Penberthy, J. S. and D. Weld: UCPOP: A Sound, Complete, Partial-Order Planner for ADL. Third International Conference on Knowledge Representation and Reasoning (KR-92), Cambridge, MA, (1992)
8. Reiter, R.: Knowledge in Action. MIT Press (2001)
9. Russell S. Norvig .P: Artificial Intelligence: A Modern Approach. Prentice Hall, (1995)
10. Sycara, K., M. Paolucci, A. Ankolekar, and N. Srinivasan: Automated discovery, interaction and composition of Semantic Web services. J. Web Semantics, **1**, Elsevier (2003) 27–46
11. Sirin, E., B. Parsia, D. Wu, J. Hendler, and D. Nau: HTN Planning for Web Service Composition Using SHOP2. J. Web Semantics, **1**, Elsevier (2004) 377–396
12. Sirin, E. and B. Parsia: Planning for Semantic Web Services. In Semantic Web Services Workshop at 3rd International Semantic Web Conference, (2004)
13. Sohrabi, S., Prokoshyna, N., McIlraith, S.A.: Web Service Composition via Generic Procedures and Customizing User Preferences. Int. Semantic Web Conf., ISWC2006 (2006) 597–611