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 being for task planning. However, no complete success of service composition by HTN is achieved as yet. The reason is the Web service composition process involves interactive dataflow between variables in precondition and input/output parameters of services. While such dataflow requires to evaluate variables in order to compose services, the performance of services is undesirable, because world-altering Web services cause to alter the world in composition processes. In this paper, instead of the HTN task planning method, we address more radical approach of HTN method for web service composition and decomposition on the premise of the openness and uncertainty of WWW. We capture composite services, which contain abstract concepts with respect to the variables of Web services, as abstract programs to be tailored to individual users and to be instantiated to executable programs in which every variable can ground in execution. 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 structural workflow or a prototype of actual programs. We formalize web service composition/decomposition by HTN method using the idea of satisfiability of situation calculus, and address the algorithm for Web service (de)composition that does not require to perform services.

1 Introduction

The Hierarchical Task Network (HTN) planning method is conceived of as a useful method for web service composition, 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 precondition and effects, and the interaction between I/O parameters and precondition/effects may happen. Second, the constraint in web service composition is not partial orders of tasks but control constructs in which there are dataflows and control flows between subtasks. Therefore, the semantics of web service composition are much more complex and analogous to programming rather than planning. The realization of the service composition ought to differ from that of task planning.

Sirin, et al. [10] achieved the web service composition using the HTN task planning method. They invented the translator from the OWL-S service description to the SHOP2 [6] task planning domain. 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 did not maintain world-altering and 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. In case that a variable in a precondition is unified to some output parameter of a Web service in planning, we need the value of the output. They argued that we do not want to actually alter 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 precondition 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 process and execution process in the dynamic world. In fact, the authors embrace the problem how to deal with the progression of situation in our decision-making support application [5], in which light anomalies of the rocket launch operation process in planning time may change to heavy anomalies in execution time and one control mode may progress to the successive control mode.

The authors claim that the uncertainty of WWW must be coped with by Web service agents situated in circumstances, that is known as situated planning agent [9], which monitors the plan execution, detects failures of the performance, replans the plan, and adapts the behavior to changeable situations. The behavior under the uncertainty is the common observation among animals and intelligent human beings in the real world, and we argue 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 terminologies in automated planning in state space [2] and situation calculus [8]. In Section 2, we review the formalization of task planning by HTN [2]. Then, we expand it for Web service composition. We discuss 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 algorithm, which is straightforwardly embodied of the algorithm using nondeterministic search technique with computational continuation. The algorithm is incomplete because of the open world assumption but do not compel us to perform services in Web composition process. In Section 4, we discuss the related work, and we make some concluding remarks.

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. 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 precondition of o, and $effects(o)$ is the effects of o. Both are a set of literals.

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 for m.

If an instance of 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* for s.

A task is an expression of the form $t(r_1, ..., r_k)$ like the name of operator and method, but a name symbol t of task differs from a method name symbol n, and a task has fewer parameters than the name of the corresponding 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 such as $task(m)$ on a method m is nonprimitive, but a subtask may be an operator.

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

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

Where U is a set of all task nodes in the network and C is a set of constraint such as partial task ordering and preconditions for tasks. Note that w contains only problematic front part of whole network in partial order HTN, and it evolves along with the progression of the state and the plan.

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 so that it includes not only atoms from precondition and effects but also inputs and outputs of Web service. Outputs of Web service are added as atomic formula 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 predecessor services. The data stream from an input to an output via services is called *dataflow*, and we call a data stream that streams in and out via services *IO fluent*.

Definition 4. *An atomic service is an expansion of the operator, and de ned 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. It has a same syntactic expression of the form $n(x_1, ..., x_k)$ as operator name $name(o)$, but variable symbol x_i may appear not only $precond(as)$ 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 a sequence of variables respectively, while $precond(as)$ and $effects(as)$ is a set of literals.

Definition 5. *A composite service is an expansion of the method, and de ned 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, $task(cs)$ is similar to that of HTN, but the notation of $subtasks(cs)$ 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 $r_1, ..., r_n$ is unifiable 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 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 task flows (abstract control flows) and dataflows are contained in $controlConstruct(cs)$. A composite service can contain only one control construct in definition, but a control construct can contain subtasks and sub control constructs in the specific form of various kind of control constructs, specifically, $sequence$, $ifThenElse$, $loopWhile$, etc.

We can consider various controlConstructs, 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 partial orders of subtasks. Therefore, a predecessor and the successor of tasks can be interleaved with 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 $if(ite)$ holds in the state. Optional $else(ite)$ is a subtask or a control construct that is performed when $if(ite)$ does not hold.

Note that $then(ite)$ is performed if and only if the condition of $if(ite)$ holds in the state, but $else(ite)$ is performed in case that not only the negation of condition $if(ite)$ holds but also it is unknown 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 holds $seq(lw)$ 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 produced from effects. On the other hand, the objective of web service composition and decomposition is to obtain a set of executable task flows or a program that is an instantiated network of web service performance in the environment, where input and output parameters are variables to be unified to constants in state, and a goal g may include output variables.

We have two categories of goals in essence in web service composition, i.e., the alteration of world by Web service performance and the information retrieval by performance of Web services. The former is the same as task planning but the latter differs from task planning in given goals. We give ground literals in atomic formula (say using individuals in OWL, like $on(A\ B)$ for Block A and Block B) as goals in task planning, but we do not designate values of service outputs as goals in web service composition. The values of output variables are the very thing we want to know in the information providing services. We are able to only designate types of variables (note that a variable is also an instance of an OWL class but unifiable to the range of instances of a class) as goal (say ?*roomtype*, a room type of hotel available). Web service decomposers must generate instantiated task flow that includes atomic services that achieve world-altering goals and information-retrieval goals. Moreover, the coupling of world-altering service performance and information-retrieval service performance may happen through variables.

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

Satisfiability of Web Service: In HTN task planning, an operator is an abstraction that stands for all instance operators named by an operator symbol. In an instance of operator o, a variable symbol in $name(o)$, $precond(o)$, and $effects(o)$ is substituted with a corresponding constant symbol. In Web service (de)composition, an atomic service can be instantiated through the substitution of all variables except IO parameters in the precondition, effects, inputs, and outputs with ground literals in the state space. In addition, IO parameters in a service must satisfy the state in the execution of the instantiated service.

To discuss the interpretation of assertions in Web service (de)composition, we consider an state transition machine. Let Σ be the state transition machine for task planning [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 is a mapping from s_{i-1} to $s^{\mathcal{I}}_{i-1}$ and s_i to $s^{\mathcal{I}}_i$, if an instance operator o in P that links from s_{i-1} to s_i has a mapping to $o^{\mathcal{I}}$ that links from $s^{\mathcal{I}}_{i-1}$ to $s^{\mathcal{I}}_i$, it is called *satis able* with respect to the operator o.

We expand this interpretation for task planning to Web service (de)composition. In case that there is a mapping from s_{i-1} to $s^{\mathcal{I}}_{i-1}$ and s_i to $s^{\mathcal{I}}_i$, 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 $s^{\mathcal{I}}_{i-1}$ to $s^{\mathcal{I}}_{i}$, we call it *satis able* with respect to the atomic web service as.

For an atomic service as, iff the precondition $precond(as)$ is satisfiable in s, namely a interpretation of condition of $precond(as)$ holds in an interpretation $s^{\mathcal{I}}_{i}$, and inputs $inputs(as)$ is satisfiable in s, namely we can find the unification for each variable of inputs onto s and s has a mapping $s^{\mathcal{I}}_{i}$, where the unification is identical to that for the instantiation of $precond(as)$, then the atomic service as is satisfiable with the respect to 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 an applicable composite service may have no model on s.

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

Satisfiability of Task: In case that a task $t(r_1, ..., r_n)$ is primitive, the task $t(r_1, ..., r_n)$ is applicable and satisfiable, iff the corresponding atomic service as which may be partially instantiated is satisfiable.

$$s \models t \Leftrightarrow (name(as) \equiv n(x_1, ..., x_n) = \sigma(t(r_1, ..., r_n), \theta)) \wedge (s \models \sigma(as, \theta))$$

Where $\sigma(t(r_1, ..., r_n), \theta)$ expresses the substitution of $t(r_1, ..., r_n)$ by unifier θ for s. Note that θ contains the accumulation of the past unification in (de)composition process.

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

$$s \models t \Leftrightarrow (name(cs) \equiv n(x_1, ..., x_k) \simeq \sigma(t(r_1, ..., r_n), \theta)) \wedge (s \models \sigma(cs, \theta))$$

Where $k \geq n$ and $\simeq$ expresses the equality of name $n = \sigma(t)$ and $x_i = \sigma(r_j)$ when $k = n$ but some of parameters x_i may not be unified to r_j when $k > n$.

Progression in Web Service Composition Let consider a progress by task t for state s_{i-1}. It seems to be the same as the expression on action $s_i = do(a, s_{i-1})$ by Reiter [8] whereas a is an action that contain only preconditions and effects. The application of task t under the state s_{i-1} yields the state s_i. However, we cannot really execute the task in service (de)composition processes, if the task is a world-altering task. Then, we cannot know values of IO parameters. Instead, we make a progression by the unification that change abstract types of variables to more special types. The progression in web service (de)composition by unification is expressed as below.

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

Satisfiability of *sequence* Control Construct: Let be *seq* a *sequence*, and let s_0 the initial state for *sequence seq*. Let us express the performance 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 $elt_1(seq)$ by making a progress of unification.

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

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, and so forth. Please note that here we omit the substitution of each element by the unifier that accumulates unifications according to the progress. Namely, $elt_2(seq)$ is $\sigma(elt_2(seq), \theta)$ exactly.

$$s_2 = \gamma(s_1, elt_2(seq)(in^1_1, ..., in^l_1))$$

We cannot say that if all tasks $elt_i(seq)(in^1_{i-1}, ..., in^k_{i-1})$ 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 progress of control constructs, we call the control constructs has a model. Thus,

$$s_0 \models seq \Leftrightarrow s_0 \models dataflow(seq) \bigwedge_{i=1 \quad k} s_{i-1} \models elt_i(seq)$$

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 the computation of satisfiability terminates, because the composite service is decomposed down to atomic services in an acyclic task network and the satisfiability computation for every atomic service terminates.

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* with respect to the condition.

– For positive condition, we have

$$s \models ite \Leftarrow (s \models if(ite) \ \wedge s \models then(ite)).$$

– For negative condition, we have

$$s \models ite \Leftarrow (s \models \neg\ if(ite)\ \wedge s \models else(ite)).$$

– For unknown condition, we have

$$s \not\models ite \Leftarrow (s \not\models if(ite)\ \wedge s \not\models \neg\ if(ite)).$$

If $if(ite)$ holds and $then(ite)$ is satisfiable for s, 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 whether ite is satisfiable. In the execution process, the value of $if(ite)$ is usually known as a result of the service execution and the progression of state, but it may be unknown in composition processes without the service execution. Therefore, the decomposer may not proceed the reasoning at this branch possibility of control. In such a case, an agent may select one of two strategies, i.e., *speculative strategy* or *assurance strategy.* In the speculative strategy, the agent aggressively takes one of the possibilities of branches, and the soundness of the instantiated program is not guaranteed. If the executer fails the execution of the generated program, the agent repairs the failure and replans at the point. In the assurance strategy, the agent defensively carries the incomplete programs to the execution phase and executes it up to the undecomposed point. The agent restarts to plan 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.

Satisfiability of *loopWhile* Control Construct: Let be s_0 an initial state for an instance of $loopWhile$, lw. When a while-condition $while(lw)$ of lw does not hold by 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, we treat it in the same way as $ifThenElse$.

As a result of one round of iteration for 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 the states may evolve on the satisfiability check for even the same procedure, because types of variables evolve more precisely step by step using the typed unification, which is described later. In the worst case, the evolution stops even if $while(lw)$ holds, thus the decomposer must detect no progression in the iteration and exit from the loop.

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 variable bindings that make a control construct in hand satisfiable, and search all possibilities of progression by substituting all variables in service parameters. Here note that input and output parameters are typed in DL or OWL, and variables in precondition and effects also typed. Therefore we need typed unification to compute satisfiability. The typed unification algorithm is described in the next section.

Let us call an instance of atomic service *ground*, if it contains ground atoms except unground variables that are input and output variables in some atomic services. In HTN method for task planning, a ground instance of operator that is applicable to s is an active candidate for the plan solution. There is no unground variable in ground operators. In HTN method for service (de)composition, we have input and output variables in ground atomic services. Therefore, the active candidate for the plan solution must be a pair of ground atomic service and its unifier that instantiated the atomic service.

Suppose that a task node u in U, $u \in U$, is in the network w. Here $task(as) = t_u$ or $task(cs) = t_u$. When $w = \langle U, CC \rangle$ is primitive, if U is grounded and CC is satisfiable for s, then w is a solution for s such that the executer can execute CC for s. Then as instantiates u so as to produce the instantiated task network w' from w. If $w = \langle U, CC \rangle$ is nonprimitive, then w is a solution for s if we can find a satisfiable unification that satisfies CC and a primitive task network w' is obtained as a result of decomposition of cs. In other words, the problem solving of service composition is to find the path of evolution of partial network w by decomposition for the subtasks of cs in the unification.

The algorithm of HTN web service composition and decomposition is described as follows. Where s is a state in a situation, w is a part of whole task network that is to be instantiated. w' is a part of task network that is instantiated. The initial input of w is a network that includes only a top task, and w' is null set. AS is a set of atomic services, and CS is a set of composite services. D is a domain knowledge of the target field.

```
**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**
    $active \leftarrow \{\langle \sigma(as), \theta \rangle \mid as =$ discover$(t_u, w, AS, D)$ and $as$ is satisfiable in $s$
                          $\theta$ is a unifier with satisfiable bindings of $as$
                          $\sigma(as)$ is a substitution of $as$ with $\theta\}$
    **if** $active = \emptyset$ **then return** fail()
    nondeterministically choose any $\langle \sigma(as), \theta \rangle \in active$
    SWHTN($\gamma(s, as), w' + \{\sigma(u)\}, \sigma(w - \{u\}), AS, CS, D$)  ; { } means a network.
  **else** ; ; $t_u$ *is nonprimitive.*
    $active \leftarrow \{\langle \sigma(cs), \theta \rangle \mid cs =$ discover$(t_u, w, CS, D)$ and $cs$ is satisfiable in $s$
                          $\theta$ is a unifier with satisfiable bindings of $cs$
                          $\sigma(cs)$ is a substitution of $cs$ with $\theta\}$
    **if** $active = \emptyset$ **then return** fail()
    nondeterministically choose any $\langle \sigma(cs), \theta \rangle \in active$
    **return** SWHTN($\sigma(s), \sigma(w' + \{u\}), \sigma(w - \{u\} + sub\{u\}), AS, CS, D$)
```

$\gamma(s, as)$ is a progression by an atomic service as. For a nonprimitive service, this algorithm decompose it into subtask nodes and evolves the state by the satisfiable unifier. For a primitive service, this algorithm makes a progression of state and accumulates the instantiated network. This algorithm contains the loop of SWHTN() via the 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 *satis able*. We used it instead of the terminology of *ground action* in HTN task planning for collecting *active* atomic service. The satisfiability checking, which deeply searches down to atomic services from a composite service, returns several unifier that satisfy the node to the state s.

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 *speculative strategy* or *assurance strategy*.

3 Implementation

3.1 State Space and Variables

Generally, an atom can be expressed as a form $p(r_1, ..., r_{n-1}, r_n)$. If an atom is a state variable such that the combination of predicate p and variables $r_1, ..., r_{n-1}$ has a mapping to r_n on each state s, it can be expressed as $p(r_1, ..., r_{n-1}) = r_n$. A state variable can be also expressed as $p(r_1, ..., r_{n-1}, r_n) = true$. Then, a negation of atom $\neg p(r_1, ..., r_{n-1}, r_n)$ can be expressed as $p(r_1, ..., r_{n-1}, r_n) = false$. On the close world assumption, the absence of positive atom means the negation of the assertion. On the open world assumption, the absence of positive and negative assertion means unknown on the assertion.

The state is expressed as a list of state variables as atom. In Lisp, the following shows an example of the state in which an individual `Lucy` has an appointment, and `HAL` has also an appointment. 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 make a typed variable in the following form in our system.

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

If `Person` is defined as class in OWL and `hasAppointment` is defined as an object property with the domain constraint `Appointment`, the system can create `?p` as an instance of `Person` and `?appt` as an instance of `Appointment`.

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 classes and individuals. A variable is also an individual of a class in the domain. Consider the following unification algorithm, where variablep(x) tests for not lisp symbols but OWL entities whether x is an OWL entities for variable, and variable?(x) tests for lisp symbols whether x is a variable. Note that a variable symbol and a constant symbol are bound to an OWL entity in our system, then a lisp symbol is unified to a lisp symbol in semantics of OWL as shown later.

```
**function** Typed-Unify($x, y, \theta$)
  **if** $\theta =$ failure **then return** failure
  **else if** $x = y$ in semantics of OWL **then return** $\theta$
  **else if** $x \neq y$ in semantics of OWL **then return** failure
  **else if** variablep($x$) **then return** Typed-Unify-Var+($x, y, \theta$)
  **else if** variablep($y$) **then return** Typed-Unify-Var+($y, x, \theta$)
  **else if** variable?($x$) **then return** Typed-Unify-Var($x, y, \theta$)
  **else if** variable?($y$) **then return** Typed-Unify-Var($y, x, \theta$)
  **else if** compound?($x$) and compound?($y$) **then**
    **return** Typed-Unify(Args($x$), Args($y$), Typed-Unify(Op($x$), Op($y$), $\theta$))
  **else if** list?($x$) and list?($y$) **then**
    **return** Typed-Unify(Rest($x$), Rest($y$), Typed-Unify(First($x$), First($y$), $\theta$))
  **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($var, x, \theta$)
  **if** $\{var/val\} \in \theta$
    **then return** Typed-Unify($val, x, \theta$)
  **else if** $\{x/val\} \in \theta$
    **then return** Typed-Unify($var, val, \theta$)
  **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 practice, our system is built on top of SWCLOS, which is an OWL Full reasoner and language [4]. In SWCLOS, every OWL entity is an object in CLOS. In the integration of our (de)composition system to SWCLOS, a variable *var* in Typed-Unify-Var+ is an object typed to OWL classes in the domain, while x may be an individual object of domain classes or may be an variable object typed to a domain class.

```
**function** Typed-Unify-Var+($var, x, \theta$)
  **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
```

Where individual() creates an individual object of the parameter. Through this 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. Generally speaking, it is valuable to give rich information of negation, disjunction, and complement in ontologies for the open world.

3.3 Nondeterministic Choice by Continuation

The computational continuation is well known as program control technique in Scheme language. 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 implement our (de)composer, 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 Toward Reasoning in Services from Reasoning in Action

The study on task planning has a long history. Recently, Ghallab, et al. [2] published a comprehensive text on automated planning of action. Reiter [8] enlightened on task planning problem from situation calculus. From the advent of Semantic Webs, Web Services plus Semantic Webs has emerged as a new field in planning, and many efforts has been made in various approaches. Berardi et al. [1] discussed the synthesis of Web services from situation calculus, but the work still stays at the closed world assumption. Sohrabi et al. [12] demonstrated the 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 by SHOP2 [10], seems to be left still open.

All of works mentioned above strongly stick the soundness and completeness of service composition. However, the authors argue that the openness and uncertainty of WWW lead us to the incompleteness when we consider the execution process. 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 (de)composition. We also suggested that we need situated planning agent that adaptively behaves in use under the incomplete service composition and the uncertainty of WWW with the premise of the open world assumption.

Sirin and Parsia [11] 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 on variable bindings. The authors' 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 correctly on the existential quantifier. On the other hand, the problem on the creation of anonymous individuals mentioned by them 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.

4.2 Web Service Composition and Decomposition

In this paper, the terminologies of both composition and decomposition, and occasionally composition/decomposition and (de)composition are used. Usually, HTN is conceived as a method for web service composition. However, the composition process in HTN is strictly coupled with the decomposition process. Ghallab, et al. often used the terminology of *decomposition tree* of HTN in their textbook [2]. We consider an agent in which the composer composes Web services in coarse grain size from scratch. In this service composition, the partial order planner technique like UCPOP [7] may be useful rather than HTN. Then, the decomposer in the agent decomposes the composed service into fine grain services by HTN. In this paper, we concentrated the discussion to HTN (de)composition process, in which we have a top task node of HTN and separated other subtasks from work flow library at first, and the top task and related abstract tasks are combined and instantiated along with the reduction the ambiguity of task parameters step by step. We call this HTN planning process *service (de)composition.*

4.3 Framework of Web Service Agent

The agent system includes an executer, memory, and user interface in addition to the composer and the decomposer [5]. The executer interprets and executes the instantiated programs with invoking Web Services. The machinery of memory works as memory for various internal data of agent. 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.

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. 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
11. Sirin, E. and B. Parsia: Planning for Semantic Web Services. In Semantic Web Services Workshop at 3rd International Semantic Web Conference, (2004)
12. 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