Executing Agent Plans by Reducing to Workflows

Tayfun Gokmen Halac, Övünç Çetin, Erdem Eser Ekinci
Rıza Cenk Erdur, Oguz Dikenelli
Ege University, Department Of Computer Engineering
35100 Bornova, Izmir, Turkey
Email: {tayfunhalac,ovunccetin,erdemeserekinci}@gmail.com
{cenk.erdur,oguz.dikenelli}@ege.edu.tr

***Abstract*—In this paper, we introduce an agent planner architecture that can reduce the basic artifacts of agent planning paradigms, semantic services and business process languages into a common workflow model. These artifacts are then executed by means of a workflow component that the architecture includes. By having a workflow component in an agent infrastructure, various agent programming paradigms including different planning approaches as well as different workflow definition languages can be executed on the same agent platform. To illustrate our ideas, we focus on the reduction of plans to the workflow model. To explicate the reduction mechanism, we have preferred to use HTN which is a widely known planning approach in multi-agent domain. Based on the semantics that we have defined for our workflow and HTN models, we have given an algorithm for transformation from HTN to workflow model.**

I. Introduction

Agents can execute various task structures in order to achieve their goals. These task structures may be components of a plan (e.g. actions), services including semantically defined web services, or workflows which are represented using an appropriate formalism such as BPEL[1], XPDL[2]. An agent may execute each of these task structures in a way that is independent of others as it is the case for an agent that can execute only plans, only OWL-S service definitions or only workflows.

On the other hand, it is usually a desired property for an agent to execute several task structures in a combined manner. For example, one or more actions of a hierarchical task network (HTN)[3], [4] plan may need to call a web service or execute a pre-defined workflow. In addition, in open and collaborative multi-agent organizations where task structures can be distributed within the environment, it is required to discover, access, compose (if needed), and execute them at run-time. To support various task execution semantics both at design time and run-time, what is needed is a special agent planner architecture that should ideally provide a unique and common basis for the execution of different task structures in a both independent and combined manner.

There are three basic requirements to support various task execution semantics in an agent architecture. First, meta-models for the representation of various task semantics are needed. OWL-S, which is a standard for defining web services semantically, and workflow definition formalisms such as BPEL are examples for such meta-models. As another example, agent plans can be represented using OWL ontologies at the meta-level. Second, a common model that will form a common execution basis for the tasks that have different semantics is needed. Based on the fact that a plan can be represented as a directed graph which can be executed as a workflow, defining a generic workflow graph model will satisfy the requirement for a common model. Finally, algorithms for the transformations from the meta-models into the common representation model should be developed.

In this paper, we introduce a planner architecture that fulfills the requirements given above. The introduced architecture includes a generic workflow graph model into which various task semantics can be transformed. This generic workflow graph model has been defined based on the abstract definition given in [5]. Within the planner architecture, we have also implemented an executor component which is used to execute the instances of the generic workflow graph model.

In literature, there are studies that aim to execute web services or workflows within a planner or an agent architecture. [6] describes how SHOP2 HTN planning system can be used to execute OWL-S descriptions. The SHOP2 planner takes the composite process defined using OWL-S as input and executes this composite process. WADE[7] is a software platform which is built on top of the well-known agent framework JADE[8]. WADE uses a directly executable simple workflow structure based on java class instead semantically described planning paradigms. Our study differs from these works, since our proposed planner architecture can support combinations of various task semantics both at design-time and run-time by providing a common execution layer for all of them. Neither [6] nor [7] aims to support more than one task execution semantics at the same time. Another point that needs attention is that the workflow graph model which constitutes the core of our common execution model is not related with the concept of executing a workflow within an agent. The workflow graph model is a directed graph structure into which various task semantics are transformed before execution.

We have implemented the planner architecture within SEAGENT[9], which is a semantic web enabled multi-agent system framework developed by our research group[10]. The planner can reduce the plans defined using OWL ontologies and OWL-S service definitions into the common workflow graph model, and then execute them. To illustrate the reduction process, we just focus on the transformation of HTN semantics into the common workflow graph model in this paper. We have chosen HTN because HTN planning is a well-known approach

that has affected the agent domain most, and has been directly used in several agent development frameworks[11], [12]. SEAGENT also incorporates HTN as its planning paradigm.

Remaining parts are organized as follows: Before giving the details of the reduction mechanism, we introduce current architecture of SEAGENT planner in Section II. In section III, details of our planner's internal workflow structure, to which HTN plans and other process definition languages are reduced, are given. Soon after, we define our enhanced HTN semantics in section IV. In section V, the algorithm that achieves the reduction from HTN to Workflow model is given, and correctness of the reduction mechanism is discussed. Section VI includes a working example and Section VII the conclusion.

II. The Architecture Of The Planner

The implementation of the proposed planner architecture is presented in Figure-1. As indicated in the figure, two vertical layers compose the overall architecture. The front layer, called Semantic Web Layer, uses ontological descriptions to represent the artifacts of the planning process. These ontology descriptions are: *Goal, Role, HTN*[1] and *OWL-S*. The goal ontology defines the objectives which the agent intends to reach within the organization. It specifies abstract definitions for agent behaviors. The role ontology, on the other hand, puts the related goal definitions together within a role description. In addition, it also defines some constraints about the considered role. Roles and goals take in charge during planning process, and then the output plan is reduced to the workflow model. Our main interest is on reduction of plans, not on this plan decision process. The HTN ontology is used to describe the agent plans using HTN planning technique. Finally, OWL-S (Web Ontology Language for Services) is a part of our proposed architecture to support semantic services. As is seen in the figure, the planner, which composes the Execution Layer, consists of four modules: *WorkflowElements*, *WorkflowEngine*, *Matchers*, *Reducers*.

The WorkflowElements module contains building blocks of our graph structure which is used to represent the agent acts as a workflow at execution time. The graph structure is based on tasks, coordinators and flows, all of which are discussed in detail in Section III.

The WorkflowEngine module is responsible for performing a workflow instance and coordinating its execution. It consists of three submodules: *ExecutionToken, LazyBinder,* and *GraphVerifier.* The ExecutionToken is the major component for the execution which traverses the nodes of the workflow instance and executes tasks of workflow instance. The LazyBinder module was developed to support dynamic graph binding. It has a special abstract graph node called LazyTask which loads the concrete graph at runtime. This dynamic binding capability makes our workflow dynamically extendable. Thus, at run-time new goals and tasks can be appended based on the state of the environment and/or agent's knowledge. Finally, the GraphVerifier module is responsible for verification of a workflow instance before it is given to the execution. It verifies the syntactical structure of the workflow and data flow over this workflow instance.

Reducers, Matchers, and *JENA*[2] form a bridge connecting the execution layer to the Semantic Web layer. The ontological definitions of the agent's actions from the Semantic Web layer are read here, and converted to the corresponding workflows. The Reducers sub-package contains the graph reducer components (*GoalReducer, HTNReducer, OWLSReducer*) that parse Goal, HTN and OWL-S definitions and reduce them into the workflow. The other module, called Matchers, contains three submodules: *RoleMatcher, GoalMatcher* and *ActMatcher.* Role and goal matchers collaborate to find an appropriate goal description for an objective which the agent must fulfill. If a goal is matched with an objective, the description of the goal is given to the GoalReducer to reduce it into a goal workflow. During the execution of the goal workflow, an ActMatcher is invoked for each sub-goal to find an appropriate act (HTN plan or OWLS service) that accomplishes the sub-goal. After the ActMatcher returns the corresponding act description, a reducer (HTNReducer, or OWLSReducer) is selected to reduce the act description to the workflow.

In this paper, we only focus on the reduction of HTN plans to workflows. To specify our reducing process, workflow and HTN semantics are formally defined in the following sections. Next, reducer algorithm is illustrated in the section V.

III. Semantics of SEAGENT Workflow Graph

As mentioned in the introduction, workflow technology has been extensively researched in the academy and as a result of these efforts this technology has reached to a high degree of maturity. Widely acceptance of workflows in industrial settings and standardization of workflow definition languages such as BPEL, XPDL are the signs of this maturity degree. On the other hand, from the execution perspective, several approaches have raised up such as executing the workflows on Petri-Nets[13] and on conceptual directed graphs[5].

Sadiq and Orlowska abstractly introduced the basic building blocks of a workflow process using the graphical terms such as nodes and flows in [5]. They also defined the basic workflow constructs such as sequence, choice, concurrency and iteration. Besides the modeling workflows, they touch on reliability of the workflow model in [14], [15]. For this purpose, some structural conflicts as deadlock, lack of synchronization, live-lock are determined. Our workflow model is built by deriving the abstract definitions of Sadiq et al. and extended by constraints to avoid structural conflicts articulated in [14].

In this section, we semantically declare the concepts and their constraints of our workflow implemetation. Before giving semantics of our workflow model, we evoke to the reader that a workflow is also a kind of directed graph[16], [17]. So, we start to explain semantics of our model by giving the formalism of the directed graph with the following definition.

Definition 1: Given directed graph is a tuple of $g = \langle V, E \rangle$, $V = \{v_0, v_1, \ldots, v_n\}$ and $E = \{\langle v_s, v_t \rangle : v_s, v_t \in V\}$.

[1] http://etmen.ege.edu.tr/etmen/ontologies/HTNOntology-2.1.4.owl

[2] We use JENA (http://jena.sourceforge.net/) to read and manipulate the ontology documents in the knowledge-base and over the Internet.

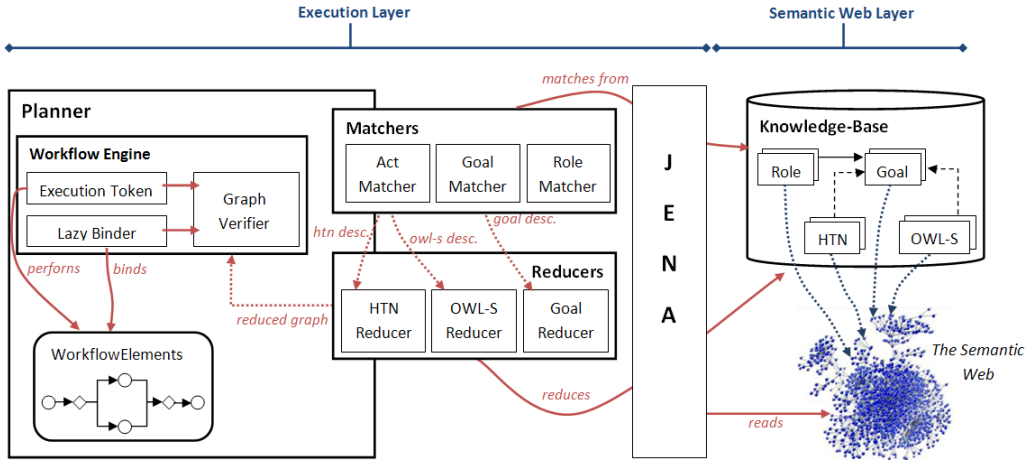

Figure 1. SEAGENT Planner Architecture

The directed graph represented with g consists of vertices V and directed edges E. A vertex, $v \in V$, specifies a node in the graph. The words, *node* and *vertex,* will be used interchangeably. A directed edge, $e \in E$, shows a directed link between two vertices of the graph. In the definition, vertex "v_s" represents the source vertex of the edge and vertex "v_t" is for the target. We define a function, *path,* that helps to gather the directed ways between two given vertices:

- $path\,(v_i, v_k) = (e_0, \ldots, e_n)$ where $v_i \in V$ and $v_k \in V$ represents the first and the last vertex of the path. *path* defines one of the directed ways between vertices v_i and v_k. The first term of the finite sequence of edges is e_0 where $source\,(e_0) = v_i$ and the last term is e_n where $target\,(e_n) = v_k$. For all terms of the sequence, target node of an edge equals to source node of the next term, $(target\,(e_0) = source\,(e_1)) \wedge \ldots \wedge (target\,(e_{n-1}) = source\,(e_n))$.
- $paths\,(v_i, v_k) = \{path_1\,(v_i, v_k), path_2\,(v_i, v_k), \ldots\}$ represents all different ways between the given two nodes.

This definition uses two functions for each edge $e \in E$: $source\,(e) = v_m$ where $v_m \in V$ represents the source vertex of e and $target\,(e) = v_n$ where $v_n \in V$ represents the target vertex of e.

Semantics of the workflow graph model, which extends the directed graph represented formally above, is defined below by giving details of the model's building blocks.

Definition 2: wfg, which is a tuple $\langle T$,C,CF, DF,IC_{wfg}, OC_{wfg},$TN\rangle$, expresses a *workflow graph* that consists of set of tasks *T*, set of flows *CF* and *DF* which represent control flow and data flow sets respectively, input and output containers IC_{wfg} and OC_{wfg}, set of coordinators C, and set of terminal nodes TN.

The workflow graph as mentioned previously is derived from the directed graph. So, when looked through the directed graph perspective, some entities of workflow graph, such as tasks, coordinators and terminal nodes, are sub-sets of the vertex set: $T, C, TN \subset V$. Also, CF and DF are specialized entities of workflow that are sub-sets of directed edge set: $CF, DF \subset E$.

Definition 3: An element of *task* set is formally defined as $\tau_i = \langle n_{\tau_i}, IC_{\tau_i}, OC_{\tau_i}\rangle \in T$. In detail, n_{τ_i} is the identifier of the task, while IC_{τ_i} and OC_{τ_i} correspond to input and output containers respectively.

- A data container, κ, is the set of data items which are needed to execute a task or generated by the task, κ = $\{d_1, d_2, \ldots, d_i\}$. *IC* and *OC* are sub types of data container, $IC, OC \subset \kappa$.
- A data item, $d_i = \langle n_{d_i}, type_{d_i}\rangle$, stands for data which is held by input and output containers. d_i is identified by its name (n_{d_i}) within the container and $type_{d_i}$ property specifies the data-type of d_i.

The required coordination of tasks of workflow and data sharing between tasks are provided by special components called *flow*s. There are two types of flows in our model: *control flows* and *data flows*. The details of the flows are determined with the following definitions.

Definition 4: A *data flow*, $df_i = \langle \tau_{src}, d_{src}, \tau_{trg}, d_{trg}\rangle \in DF$ where $\tau_{src}, \tau_{trg} \in T$, $d_{src} \in OC_{\tau_{src}}$, $d_{trg} \in IC_{\tau_{trg}}$, is a type of flow that is used for data migration between two tasks. It transmits the value of the source task(τ_{src})'s output item (d_{src}) to the target task(τ_{trg})'s input item (d_{trg}) after the source task performed.

Data flows are important to supply inputs to the tasks. So, to supply the inputs safely, we define some constraints on data flows. Before declaring these constraints, we define two supportive functions, $inData$ and $outData$, as below:

$outData\,(d_{src}) = \{df : df \in DF \wedge sourceItem\,(df) = d_{src}\}$ where d_{src} is a data item, returns the set of data flows whose source data item is d_{src}.

$inData\,(d_{trg}) = \{df : df \in DF \wedge targetItem\,(df) = d_{trg}\}$ where d_{trg} is a data item. It returns the set of data flows whose target is d_{trg}.

$sourceItem$ and $targetItem$ functions are similar to $source$ and $target$ functions, but they are used to retrieve the data items bound by the specified data flow.

Now, we can describe the constraints on data flows using these functions:

- **(C)** There should not be more than one data flow between any two data items: $\forall d_{src}, d_{trg}\ (|outData\,(d_{src}) \cap inData\,(d_{trg})| \leq 1)$

- **(C)** Data type of the target task's input item must be equal or subsume the type of the source task's output item: $\forall df \in DF\ \left(type_{d_{src}} \subseteq type_{d_{trg}}\right)$

As we mentioned above, a data flow expresses the direction of the data item migration. Differently from the data flow, on the other hand, a control flow is used to specify the execution sequence of task nodes in a workflow graph.

Definition 5: A *control flow* is a tuple, $cf_i = \langle v_{src}, v_{trg} \rangle \in CF$, consisting of source vertex (v_{src}) and target vertex (v_{trg}). Control flows are more decisive than data flows on process of workflows. To avoid inconsistencies, control flows must be constructed according to some defined constraints. Before declaring these constraints on our workflow model, we describe two supportive functions, $inControl$ and $outControl$, to make the constraints more understandable:
$inControl(n) = \{cf : cf \in CF \wedge target\,(cf) = n\}$ where $n \in V$, acquires the set of control flows whose target node is n. $outConrol(n) = \{cf : cf \in CF \wedge source\,(cf) = n\}$ where $n \in V$, returns the set of control flows whose source is n.

Now, we can specify the constraints using these functions:

- **(C)** All flows must have two different nodes connected to their two sides. $\forall f \in E\ (source\,(f) \neq target\,(f))$
- **(C)** A task have to be source or target of only one control flow. $\forall \tau \in T\ (|inControl(\tau)| = 1 \wedge |outControl(\tau)| = 1 \wedge inControl\,(\tau) \neq outControl\,(\tau))$
- **(C)** There should not be more than one direct control flow between any two nodes. $\forall v_m \in V, \forall v_n \in V\ (|outControl\,(v_n) \cap inControl\,(v_m)| \not> 1)$
- **(C)** The source node of a control flow must be ahead of the target node in the order of execution. $\forall v_m, \forall v_n\ ((path(v_m, v_n) \neq \emptyset) \rightarrow (outControl\,(v_n) \cap inControl(v_m) = \emptyset))$. As an exception, in an iteration structure, one of the outgoing flows of a choice node goes to the preceding merge node.

Although the constraints on control and data flows help to build consistent work flows, they are not enough. Control and data flows also must be compatible with each other in terms of workflow direction.

- **(C)** Since a data flow transmits the data from the source node to the target node, the source node must be finished before the target node starts. Therefore, the source node must always precede the target node within the workflow in terms of execution sequence.
  $\forall \tau_m, \forall \tau_n\ ((path\,(\tau_m, \tau_n) \neq \emptyset) \rightarrow outData\,(\tau_n) \cap (inData\,(\tau_m) = \emptyset))$
  $\forall \tau_m, \forall \tau_n\ ((outData\,(\tau_m) \cap inData\,(\tau_n) \neq \emptyset) \rightarrow (path\,(\tau_m, \tau_n) \neq \emptyset))$
  $inData(\tau) = \{df : df \in DF \wedge target(df) = \tau\}$ where $\tau \in T$, returns the set of data flows whose target is τ.
  $outData(\tau) = \{df : df \in DF \wedge source(df) = \tau\}$ where $\tau \in T$, returns the set of data flows whose source is the given task.

Data migration on the workflow and execution ordering of the tasks can be provided easily via flows. But they are not sufficient when some complex structures, which come from the nature of the workflow, such as concurrency, alternation and iteration, are considered. We need some special nodes for the purpose of implementing these structures. These special nodes named as *coordinator nodes*, will be defined next.

Definition 6: Here, we define all sub-sets of *coordinators*, *C,* together. There are four types of coordinator nodes; *choice*, *merge*, *fork*, *synchronizer*; $CH, MR, FR, SY \subset C \subset V$.

- A *choice* node, $ch_i = \langle n_{ch}, d_{cond} \rangle \in CH$, has more than one outgoing flows and contains a condition input d_{cond} to select a branch.
- A *merge* node, $mr_i \in MR$, has more than one incoming branches and it is used to join the mutually exclusive paths which are split by a choice node.
- A *fork* node, $fr_i \in FR$, has more than one outgoing flows and enables all of them at the same time.
- A *synchronizer* node, $sy_i \in SY$, has more than one incoming flows, which are activated concurrently by a fork node, and waits all to be finished. In other words, it synchronizes the concurrent execution paths within a workflow.

The coordinator nodes are used in pairs to construct exclusive choice and parallel split workflow patterns[18]. The exclusive choice pattern creates a divergence point into two or more branches such that only one of which is selected for the execution. The parallel split pattern, on the other hand, provides concurrent execution paths which are activated simultaneously. An exclusive choice is constructed with a $\langle ch_i, mr_i \rangle$ pair, while a $\langle fr_i, sy_i \rangle$ pair composes a parallel split.

As is clearly stated, the coordinator nodes are required to build workflows including complex patterns. But misusage of the coordinators may result in defected workflows. Therefore, the following constraints should be defined on the coordinator nodes to provide a consistent workflow.

- **(C)** Since the aforementioned workflow patterns are constructed using coordinator pairs, there must exist a merge node for each choice node, and a synchronizer node for each fork node. These two constraints could be expressed by $f : CH \rightarrow MR$ and $g : FR \rightarrow SY$ functions, which are one-to-one and onto, respectively.
- **(C)** All choice and fork nodes have one incoming and more than one outgoing flows. $\forall n \in FR \cup CH\ ((|inControl(n)| = 1) \wedge (|outControl(n)| > 1))$
- **(C)** All synchronizer and merge nodes have at least two incoming flows and only one outgoing flow. $\forall n \in SY \cup MR\,((|inControl(n)| > 1) \wedge (|outControl(n)| = 1))$

The node definitions made so far specify the intermediate nodes. In other words, we did not give any definition of nodes which represents the end points of the workflow up to now. The following definition, on the other hand, explains the terminal nodes, $TN = \{v_i, v_f\}$, used for this purpose.

Definition 7: Initial node, $v_i \in V$, represents the beginning of the workflow, while the final node, $v_f \in V$, represents the end, $wfg_n = (v_i, \{\tau_1, \tau_2, \ldots, \tau_n\}, \{c_1, c_2, \ldots, c_n\}, v_f)$.

- **(C)** v_i is the first node of the wfg_n, it has no incoming but only one outgoing control flow: $\forall v_i\ ((inControl\,(v_i) = \emptyset) \wedge (|outControl\,(v_i)| = 1))$
- **(C)** v_f is the last node of the wfg_n, it has only one incoming but no outgoing control flow: $\forall v_f\ ((inControl\,(v_f) = 1) \wedge (outControl\,(v_f) = \emptyset))$

- **(C)** Each workflow graph contains exactly one initial and one final node.
  $\forall wfg\ (wfg \in WFG \rightarrow wfg \ni v_i \wedge wfg \ni v_f)$

IV. Semantics of SEAGENT HTN

Previously, we gave semantics of our workflow model. Our approach, as mentioned in the introduction, is to design and implement a planner architecture that enables to execute different planning paradigms and workflow definition languages in the same agent architecture. Due to this purpose, we choose HTN paradigm, mostly used planning paradigm in the agent literature.

Semantics of HTN is firstly articulated by Kutluhan et al. in [3]. In his model, HTN is closer to AI problem solving. For the purpose of using HTN in web agent programming, Sycara et al. reformed it in [4]. They contributed the link concept to provide a unified information and control flow within plans. Although that contribution makes HTN plans more tractable, it allows designing error-prone plans. Our HTN model is a detailed revision of Sycara et al.'s that is extended by exhaustive link definitions and constraints that permit to avoid designing erroneous plans. Base concept of our HTN model is *task;*

Definition 8: An HTN *task*, $\theta_i = \langle n_{\theta_i}, P_{\theta_i}, O_{\theta_i} \rangle \in \Theta$, is generalization of the primitive task (action) and the compound task (behavior), $A \subset \Theta$, $B \subset \Theta$.

A task encapsulates the common properties of behaviors and actions, such as provisions, outcomes, and name. But they are distinguished by other properties explained below.

Definition 9: A *behavior*, $\beta_i = \langle n_{\beta_i}, P_{\beta_i}, O_{\beta_i}, ST_{\beta_i}, \Gamma_{\beta_i} \rangle \in B$, represents a compound task which encloses other tasks. A behavior (β_i) corresponds to a workflow graph (wfg) that has sub nodes which may be primitive actions or other behaviors. In the definition, a behavior consists of name, provisions, outcomes, subtasks, and subtask links respectively. n_{β_i} is a label that distinguishes the behavior from the others. Since a behavior is a compound task, it cannot be performed directly. It must be decomposed to its primitive actions whose executions contribute toward accomplishment of the behavior.

Definition 10: An *action*, $\alpha_i = \langle n_{\alpha_i}, P_{\alpha_i}, O_{\alpha_i}, \lambda_{\alpha_i} \rangle \in A$, is a primitive task that is executed directly. It corresponds an indecomposable workflow task node.

An action consists of name, provisions, outcomes, and a function. n_{α_i} is a label that distinguishes the action from the others. Because actions are executable entities within a plan, they must implement a function, λ_{α_i}, which fulfills the actual work of the action.

Definition 11: A *parameter*, $\pi_i = \langle n_{\pi_i}, type_{\pi_i} \rangle \in \Pi$, stands for data which is needed or produced by tasks. A parameter π_i consists of name (n_{π_i}) and type $(type_{\pi_i})$.

The parameter is an abstract concept that cannot be seen in an HTN plan. Parameter is generalization of provision and outcome, $P \subset \Pi$, $O \subset \Pi$. The definitions of these concepts are given below.

Definition 12: A *provision*, $p_i = \langle n_{p_i}, type_{p_i}, value_{p_i} \rangle \in P$, represents the data which is needed for execution of task. Within a workflow, P corresponds an input container and each data item in it represents a provision. Therefore, it provides the data which is required for starting the task's execution and whose value can be obtained from an outcome of the preceding task or from an external resource.

Definition 13: An *outcome*, $o_i = \langle n_{o_i}, type_{o_i}, value_{o_i} \rangle \in O$, represents the data which is produced during execution.

Some tasks gather information, which is represented by outcomes, during execution. They can be passed to needer tasks. O corresponds the output container, which consists of data items that represents outcomes, of a task within a workflow.

Definition 14: A *state* (or outcome state), $s_i \in S$, is a label on a link specifying that the branch will be executed in which condition.

State instances are used to construct branching or concurrency structures within plans. In detail, outgoing links with distinct outcome state results in an exclusive choice pattern, while the same outcome states form a parallel split.

Definition 15: $ST = \{\theta_1, \theta_2, \ldots, \theta_i\}$ indicates the set of *subtasks* of a behavior.

A constraint on subtasks is revealed below.

- **(C)** A task can be child of exactly one behavior (except the root behavior which represents the HTN plan). In other words, a task can be included by only one ST.
  $\forall \theta \in \Theta\ (\theta \in ST_{\beta_m} \rightarrow \theta \notin ST_{\beta_n})$

Up till now, we have mentioned about the HTN actions, behaviors and relation between a behavior and its subtasks. For the rest of this section, link definitions, which forms control and information flows between tasks, will be given. For that purpose, we define the link set, Γ, that is the super set of provision, order, inheritance and disinheritance links, $PL, OL, IL, DL \subset \Gamma$.

- **(C)** An important constraint on links: The source and the target task of a link must be different: $\forall l \in \Gamma\ (source\,(l) \neq target\,(l))$

Here we define two functions that are used in determining the details of links: For $\forall \theta \in \Theta$

$inLink\,(\theta) = \{link : (link \in \Gamma) \wedge (target(link) = \theta)\}$

$outLink\,(\theta) = \{link : (link \in \Gamma) \wedge (source(link) = \theta)\}$

As mentioned, there are four types of link: *provision link, order link, inheritance link* and *disinheritance link*. While provision links coincide with both data and control flows, order links correspond to control flows only. Inheritance and disinheritance links are data flows between a behavior and its subtasks. Here, the formal definitions of links and their constraints are given with necessary logical functions.

Definition 16: An *order link, $oLink_i = \langle \theta_{src}, \theta_{trg}, s \rangle \in OL$,* represents a control flow between two tasks and it designates the execution order.

Order links consist of source task, target task, and state. By using order links and states together, we can create plans including conditional branching and parallel execution paths.

- **(C)** Source and target tasks of an order link must be included in the same subtask list. In other words, an order link can connect two tasks if both are the subtask of the same behavior. $\forall link \in OL(source(link) \in ST_{\beta_n} \leftrightarrow target(link) \in ST_{\beta_n})$

- **(C)** At most one order link can be constructed between two tasks. $\forall\theta_{src}, \forall\theta_{trg}$ $(|(outLink\,(\theta_{src}) \in OL) \cap (inLink\,(\theta_{trg}) \in OL)| \leq 1)$

We define a generalized concept, *parameter link* $(\pi\mathbb{L})$,where $PL, DL, IL \subset \Pi\mathbb{L}$, for the rest of link types: *inheritance*, *disinheritance* and *provision link*s. All these links constructs data flows between the tasks by mapping the source and the target parameter of these tasks. A significant point about the parameter mapping is compatibility of the parameter types: $\forall\pi\mathbb{L} \in \Pi\mathbb{L}\,(type\,(sourceParam\,(\pi\mathbb{L})) \subseteq type\,(targetParam\,(\pi\mathbb{L})))$ where *sourceParam* and *targetParam* functions are used to retrieve the source and the target parameter of the parameter link respectively.

Two definitions below specify two supportive functions that are used to get incoming and outgoing parameter links which are used to bind the given parameter. These functions will help to describe the later definitions.

$inLink\,(\pi) = \{\pi\mathbb{L} : (\pi\mathbb{L} \in \Pi\mathbb{L}) \wedge (targetParam(\pi\mathbb{L}) = \pi)\}$ where $\forall\pi \in \Pi$.

$outLink\,(\pi) = \{\pi\mathbb{L} : (\pi\mathbb{L} \in \Pi\mathbb{L}) \wedge (sourceParam(\pi\mathbb{L}) = \pi)\}$ where $\forall\pi \in \Pi$.

Definition 17: A *provision link*, $pLink_i = \langle\theta_{src}, \theta_{trg}, o_s, p_t, s\rangle \in PL$, represents a data and a control flow between two tasks.

A provision link binds an outcome of the source task and a provision of the target task. If a conditional branching occurs after the source task, the outcome state of the link (s) maps a particular branching condition to the target task.

- **(C)** Source and target tasks of a provision link must be the child of the same behavior. $\forall pLink \in PL$ $(source\,(pLink) \in ST_{\beta_n} \leftrightarrow target\,(pLink) \in ST_{\beta_n})$
- **(C)** Source parameter of a provision link must be an outcome and target parameter must be a provision. $\forall pl \in PL$ $((sourceParam(pl) \in O) \wedge (targetParam(pl) \in P))$
- **(C)** At most one provision link can be constructed between the same outcome-provision pair. $\forall o_{src}, \forall p_{trg}$ $(|\,(outLink\,(o_{src}) \cap PL) \cap (inLink\,(p_{trg}) \cap PL)\,| \leq 1)$
- **(C)** Either an order link or a provision link can be constructed between two tasks.

$$\forall\theta_{src}, \forall\theta_{trg}\,((outLink\,(\theta_{src}) \cap inLink\,(\theta_{trg}) \cap OL \neq \emptyset) \rightarrow (outLink\,(\theta_{src}) \cap inLink\,(\theta_{trg}) \cap PL = \emptyset))$$

$$\forall\theta_{src}, \forall\theta_{trg}\,((outLink\,(\theta_{src}) \cap inLink\,(\theta_{trg}) \cap PL \neq \emptyset) \rightarrow (outLink\,(\theta_{src}) \cap inLink\,(\theta_{trg}) \cap OL = \emptyset))$$

Definition 18: An *inheritance link*, $iLink_i = \langle\beta_{src}, \theta_{trg}, p_s, p_t\rangle \in IL$, represents a parameter link between a behavior and one of its subtasks. It corresponds to a data flow from the initial node of a workflow to a subtask.

Inheritance link consists of source behavior, target task, a provision of source behavior, and a provision of target sub task. $\beta_{src} \in B$ and $\theta_{trg} \in ST_{b_{src}}$.

- **(C)** Source and target parameter of an inheritance link must be a provision. $\forall il \in IL$ $((sourceParam\,(il) \in P) \wedge (targetParam\,(il) \in P))$
- **(C)** At most one inheritance link can be constructed between the same provision pairs. $\forall p_{src}, \forall p_{trg}$ $(|outLink\,(p_{src}) \cap inLink\,(p_{trg}) \cap IL| \leq 1)$
- **(C)** Each provision of the root behavior must be bound with at least one inheritance link. $\forall p \in P_{\beta_{root}}$ $(|(outLink\,(p) \cap IL)| > 0)$ where $\beta_{root} \in B$.

Definition 19: A *disinheritance link*, $dLink_i = \langle\theta_{src}, \beta_{trg}, o_s, o_t\rangle \in DL$, represents a parameter transfer from a task to parent behavior. It corresponds to a data flow from a subtask to the final node of a workflow.

Disinheritance link consists of source task, target behavior, an outcome of source sub task, and an outcome of target behavior. Source task of a disinheritance is child of the target task, $\beta_{trg} \in B$ and $\theta_{src} \in ST_{\beta_{trg}}$.

- **(C)** Source and target parameter of a disinheritance link must be an outcome. $\forall dl \in DL$ $((sourceParam(dl) \in O) \wedge (targetParam(dl) \in O))$
- **(C)** At most one disinheritance link can be constructed between the same outcome pairs. $\forall o_{src}, \forall o_{trg}$ $(|outLink\,(o_{src}) \cap inLink\,(o_{trg}) \cap DL| \leq 1)$
- **(C)** A disinheritance link must be provided for each outcome of a behavior to collect all outcomes from the sub-tasks. If there is an exclusive choice structure, a disinheritance link must be constructed for all exclusive paths to fulfill all outcomes of the behavior. $\forall\beta_i \in B\,(\forall o_n \in O_{\beta_i}\ (|inLink\,(o_n) \cap DL| > 1))$

V. Transformation of HTN into Workflow

To implement our approach about executing different planning paradigms and workflow definition languages in the same agent architecture, *HTNReducer*, which is a component of the Reducers submodule as mentioned in Section II, is used to transform an HTN definition into a workflow before execution. In this section, this transformation process is introduced within the scope of our workflow-based HTN planner.

Algorithm 1 Reduction of an HTN behavior to a workflow

Input: an HTN behavior β.
Output: a workflow wfg.

1) Initiate a workflow graph wfg corresponding to β.
2) Create the nodes corresponding to the subtasks of β.
   a) If subtask is an HTN behavior, then apply the same process from step 1 and create a complete subworkflow for the subbehavior.
   b) For an HTN action, otherwise, create a primitive workflow task node.
3) Construct flows between workflow tasks in wfg.
4) Put required coordinator nodes to junction points.

A. Reduction Algorithm

Based on the formal definitions in Section III and Section IV, we have developed an algorithm shown in Algorithm-1 for reducing a behavior description to the corresponding workflow. For this purpose, the HTN reduction algorithm constructs a part of the graph in a few steps. It begins the process with creating an empty graph, which consists of initial and final nodes, and the data containers only, for the behavior. Then, it creates workflow task nodes for the subtasks of the behavior

and adds them into the empty graph. After the subtask nodes are added to the graph model, the next step is constructing the flows between these nodes. Finally, the last step of the reduction process, which follows the flow construction, is placing the coordinator nodes to the required locations within the graph. The steps of the algorithm are elaborated below.

In step 2, a node is created for each subtask of the given behavior according to its type. If the subtask is an HTN action, a primitive task node is constructed together its data containers. Otherwise, for an HTN behavior, the reduction process is achieved for this behavior and a graph model is created. The point to take into consideration is recursion while creating subnodes of workflow. By means of this recursion a sub-behavior is reduced prior to its parent.

As previously mentioned, the next step following the subtask creation is connecting the flows between these nodes. To do this, appropriate flow(s) for each link that is defined by the behavior description is constructed. The link type specifies the flow type and end nodes of the flow. For an inheritance link, a data flow from the initial node of the graph (i_{wfg}) to the target task of the link is constructed. A disinheritance link corresponds to a data flow between the source task of the link and the final node of the graph (f_{wfg}). For order links and provision links, on the other hand, a control flow is constructed for each. In addition to the control flow, a data flow is also constructed for a provision link.

After the flow construction phase, we obtain a raw graph model that consists of only task nodes and flows between them. There is no coordination component in this model. The last step of the algorithm, in line 4, overcomes this lack by placing the coordinators to the appropriate locations within the graph. To do this, the divergence and convergence points are marked with special nodes and then these special nodes are replaced with suitable coordinator nodes.

As a result, at the end of the reduction process, we obtain a complete workflow graph corresponding to the given HTN behavior. The graph contains task nodes that are connected with the flow objects, and the coordinator nodes that determine the flow of execution.

The outputs of our algorithm for primitive HTN patterns are represented in Figure-2. These patterns are composed of behaviors which have only primitive subtasks and other building blocks of HTN. Since plans, which have only primitive subtasks, can be defined by assembling these patterns, adding new actions to them and constructing new links between actions, they can be transformed into workflows.

To understand the reduction process better, we explain it by demonstrating one of primitive patterns. (see Figure-2(F)) The input behavior can be represented as $\beta_1 = \langle 'BH1', \{p_{\beta_1}\}, \{o_{\beta_1}\}, \{\alpha_1, \alpha_2, \alpha_3\}, \{pLink_1, pLink_2\}\rangle$ where $\alpha_1 = \langle 'AC1', \{p_{\alpha_1}\}, \{o_{\alpha_1}\}, \lambda_{\alpha_1}\rangle$, $\alpha_2 = \langle 'AC2', \{p_{\alpha_2}\}, \{o_{\alpha_2}\}, \lambda_{\alpha_2}\rangle$, $\alpha_3 = \langle 'AC3', \{p_{\alpha_3}\}, \{o_{\alpha_3}\}, \lambda_{\alpha_3}\rangle$ and $pLink_1 = \langle \alpha_1, \alpha_2, o_{\alpha_1}, p_{\alpha_2}, 'S_1'\rangle$, $pLink_2 = \langle \alpha_1, \alpha_3, o_{\alpha_1}, p_{\alpha_3}, 'S_1'\rangle$.

- At the start, an empty workflow $wf_1 = \langle \emptyset, \emptyset, \{\langle i_{wf_1}, f_{wf_1}\rangle\}, \emptyset, \emptyset, \emptyset, \{i_{wf_1}, f_{wf_1}\}\rangle$ is created, in line 1.
- In the next step, in line 2, the actions are converted to primitive workflow tasks and these tasks are inserted to task set: $T_{wf_1} = \{\langle 'T1', \{p_{\tau_1}\}, \{o_{\tau_1}\}\rangle, \langle 'T2', \{p_{\tau_2}\}, \{o_{\tau_2}\}\rangle, \langle 'T3', \{p_{\tau_3}\}, \{o_{\tau_3}\}\rangle\}$

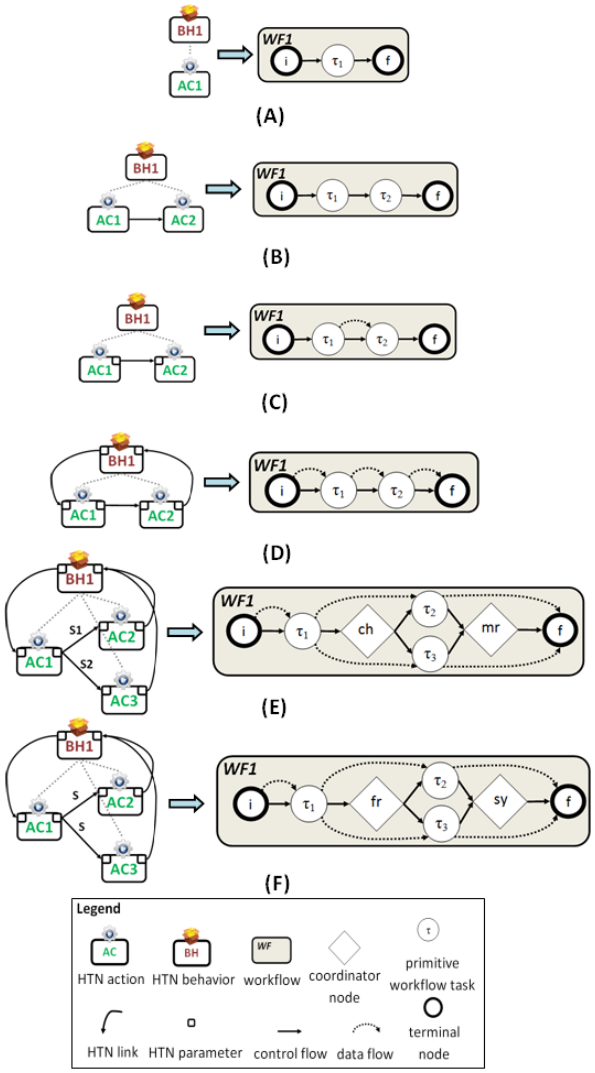

Figure 2. Reduction of Primitive HTN Patterns

- The control flow, $CF_{wf_1} = \{\langle i_{wf_1}, \tau_1\rangle, \langle \tau_1, \tau_2\rangle, \langle \tau_1, \tau_3\rangle, \langle \tau_2, f_{wf_1}\rangle, \langle \tau_3, f_{wf_1}\rangle\}$, and data flow $DF_{wf_1} = \{\langle i_{wf_1}, d_{in_{wf_1}}, \tau_1, d_{in_{\tau_1}}\rangle, \langle \tau_1, d_{out_{\tau_1}}, \tau_2, d_{in_{\tau_2}}\rangle, \langle \tau_1, d_{out_{\tau_1}}, \tau_3, d_{in_{\tau_3}}\rangle, \langle \tau_2, d_{out_{\tau_2}}, f_{wf_1}, d_{out_{wf_1}}\rangle, \langle \tau_3, d_{out_{\tau_3}}, f_{wf_1}, d_{out_{wf_1}}\rangle\}$ sets are filled in line 3.
- Finally, a fork-synchronizer node pair is inserted to required points, in line 4. This operation fills the coordinator node set $C_{wf_2} = \{fr_1, sy_1\}$ and updates the control flow set $CF_{wf_1} = \{\langle i_{wf_1}, \tau_1\rangle, \langle \tau_1, fr_1\rangle, \langle fr_1, \tau_2\rangle, \langle fr_1, \tau_3\rangle, \langle \tau_2, mr_1\rangle, \langle \tau_3, mr_1\rangle, \langle mr_1, f_{wf_1}\rangle\}$.

B. Correctness of Reduction

Theorem 1: Let β is a behavior defined with HTN semantics. $REDUCE(\beta)$ terminates and returns a workflow wf.

Proof: A behavior represents a tree, and is reduced completely after all subtasks are reduced. So, from the line 2a of algorithm, the algorithm is executed over again for subbehaviors until reaching to the leaf actions. Finally, after

the leaves are transformed in line 2b, algorithm proceeds and bottom-up construction of root behavior is achieved. ■

Theorem 2: Let $B = \{\beta_1, \beta_2, \ldots, \beta_n\}$ be a collection of HTN behaviors that conforms our constraints and β be one of these. Let $wfg = REDUCE(\beta)$, then wfg is the workflow which corresponds to behavior β.

Proof: The proof of the theorem is by induction:

Hypothesis For an HTN behavior β, there exists a workflow graph $wf = \langle T_{wf}, C_{wf}, CF_{wf}, DF_{wf}, IC_{wf}, OC_{wf}, TN_{wf}\rangle$ where T_{wf} contanins the workflow tasks corresponds to sub tasks of β, CF_{wf} and DF_{wf} contains the flows corresponds to links of HTN, and IC_{wf} and OC_{wf} contains inputs and outputs which corresponds to provisions and outcomes of β.

Base Case Suppose β is a behavior with only one action α_1 as sub task. The reduction of β ends up with a workflow $wf = \langle T_{wf}, \emptyset, CF_{wf}, \emptyset, \emptyset, \emptyset, TN_{wf}\rangle$ where $T_{wf} = \{\tau_1\}$ and $CF_{wf} = \{\langle i_{wf}, \tau_1\rangle, \langle \tau_1, f_{wf}\rangle\}$. As is seen in line 2b, after workflow is created in line 1, τ_1 is constructed for α_1 and then it is connected with i_{wf} and f_{wf} in line 3. (see Figure-2(A))

Inductive Step Besides the sequence structure in HTN, there exists a few structures such as nesting, conditional branching and parallelism. We will analyze each of the structures case by case to show that results of our translation are correct.

Case 1: While HTN plans can only be extended breadth-wise with primitive subtasks, expansion in depth is provided with subbehaviors. (see Figure-3)

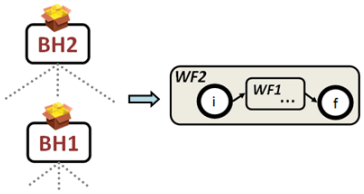

Figure 3. Reduction of nested behavior

Suppose β is a behavior with a subbehavior β_{sub}. From our hypothesis we know that there exists a workflow wf_{sub} for subbehavior β_{sub}. The reduction of β leads to a workflow $wf = \langle T_{wf}, \emptyset, CF_{wf}, \emptyset, \emptyset, \emptyset, TN_{wf}\rangle$ where $T_{wf} = \{wf_{sub}\}$ and $CF_{wf} = \{\langle i_{wf}, wf_{sub}\rangle, \langle \tau_1, wf_{sub}\rangle\}$.

Case 2: Suppose $\beta = \langle 'BH1', \emptyset, \emptyset, \{\theta_1, \theta_2, \theta_3\}, \Gamma_{\beta_1}\rangle$, where $\Gamma_{\beta_1} = \{\langle \theta_1, \theta_2, ' S_1'\rangle, \langle \theta_1, \theta_3, ' S_2'\rangle\}$, is a behavior with a conditional branching structure. (similar to Figure-2(E))
From our hypothesis we assume that we have valid workflow tasks τ_1, τ_2, τ_3 which correspond to the HTN tasks $\theta_1, \theta_2, \theta_3$. The behavior β is reduced to a workflow $wf = \langle T_{wf}, C_{wf}, CF_{wf}, \emptyset, \emptyset, \emptyset, TN_{wf}\rangle$ where $T_{wf} = \{\tau_1, \tau_2, \tau_3\}$, $C_{wf} = \{ch_1, mr_1\}$ and $CF_{wf} = \{\langle i_{wf}, \tau_1\rangle, \langle \tau_1, ch_1\rangle, \langle ch_1, \tau_2\rangle, \langle ch_1, \tau_3\rangle, \langle \tau_2, mr_1\rangle, \langle \tau_3, mr_1\rangle, \langle mr_1, f_{wf}\rangle\}$. After raw graph is obtained, choice and merge nodes are inserted to the beginning and the end of the exclusive choice structure in line 4.

Case 3: Suppose $\beta = \langle 'BH1', \emptyset, \emptyset, \{\theta_1, \theta_2, \theta_3\}, \Gamma_{\beta_1}\rangle$, where $\Gamma_{\beta_1} = \{\langle \theta_1, \theta_2, ' S_1'\rangle, \langle \theta_1, \theta_3, ' S_1'\rangle\}$, is a behavior with a parallelism structure. (similar to Figure-2(F))
From our hypothesis we know that we have corresponding workflow tasks τ_1, τ_2, τ_3 of the HTN tasks $\theta_1, \theta_2, \theta_3$. The behavior β is reduced to a workflow $wf = \langle T_{wf}, C_{wf}, CF_{wf}, \emptyset, \emptyset, \emptyset, TN_{wf}\rangle$ where $T_{wf} = \{\tau_1, \tau_2, \tau_3\}$, $C_{wf} = \{fr_1, sy_1\}$ and $CF_{wf} = \{\langle i_{wf}, \tau_1\rangle, \langle \tau_1, fr_1\rangle, \langle fr_1, \tau_2\rangle, \langle fr_1, \tau_3\rangle, \langle \tau_2, sy_1\rangle, \langle \tau_3, sy_1\rangle, \langle sy_1, f_{wf}\rangle\}$. In line 4, fork and synchronizer nodes are inserted to the beginning and the end of the parallel split structure in the raw workflow. ■
We proved that our reduction mechanism transforms the HTN plan into the workflow model correclty. In our proof, we showed the correspondence of the result workflow to the input plan.

VI. Case Study

To illustrate the reduction of plans to workflows, a tourism application is implemented as a case study with SEAGENT Framework. In this application an agent which plays tourism agency role is responsible for making a vacation plan. A plan ontology (*BHPlanVacation*) which is the implementation of planning a vacation goal (*PlanVacation*) is provided in the knowledgebase of this agent.

Ontology[3] individual of BHPlanVacation is depicted in Figure-5(B). BHPlanVacation behavior has three subtasks, and it needs location information (*location* provision) and vacation budget (*budget* provision) to make plan. After the execution of the behavior is completed, it gathers the remainder of budget (*remainder* outcome). Firstly, a hotel room is booked in specified holiday resort (*ACBookHotelRoom*), and remainder from budget is passed to the next task. After reservation operation, a travel ticket is bought according to the customer request (*BHBuyTravelTicket*). Finally, a car is rented to go to the hotel from the airport or terminal (*ACRentCar*), and the remainder value is passed to the parent behavior. Representation of the plan which is designed according to our HTN definitions in SEAGENT HTN Editor[4] is shown in Figure-5(A).

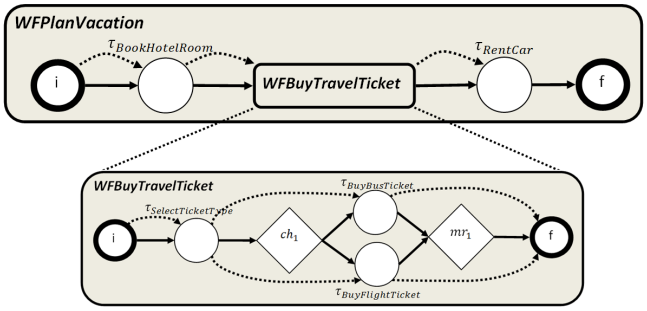

Figure 4. WFPlanVacation workflow

When agent determines to execute the PlanVacation goal, it gives the goal description to the planner. After the planner ascertains that the goal is atomic, it searches for the appropriate act in the knowledgebase via the ActMatcher. The ActMatcher finds the BHPlanVacation HTN description and

[3]Full version: http://www.seagent.ege.edu.tr/etmen/LADS009CaseStudy.zip
[4]HTN Editor is a part of the SEAGENT Development Environment that is used to build up HTN ontologies easily.

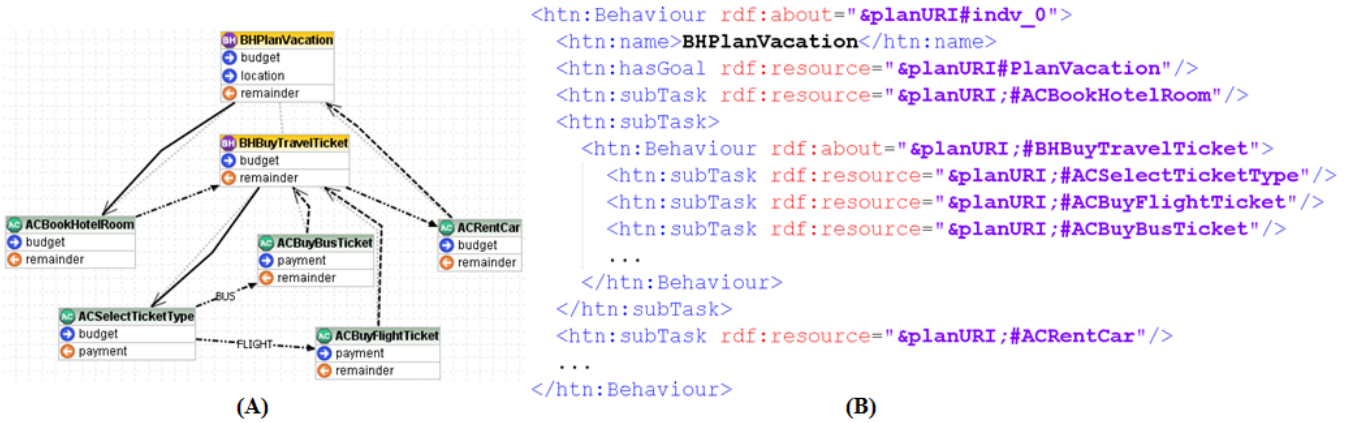

Figure 5. BHPlanVacation A) HTN Representation B) Ontology Individual

transmits it to the HTNReducer to reduce it to workflow. After the HTNReducer completes reduction, the generated workflow is the executable form of the plan description. The workflow which is constructed by HTNReducer is shown in Figure-4.

As is seen in the Figure-4, the workflow tasks are created for all actions of plan and the subbehavior is converted to subworkflow. After the workflow that corresponds to BHPlan-Vacation is constructed, the planner starts to proceed on the workflow via the ExecutionToken. Tasks are performed when the token visits them. Execution of a task means execution of the java class that is attached to the corresponding HTN action. Java reflection API is used to create and execute action class instances.

VII. Conclusion

This paper briefly depicts the architecture of SEAGENT agent development framework's planner. The main characteristic of the proposed architecture is its being based on the workflow technology and its ability to process the artifacts of agent programming paradigms such as plans, services, goals, and roles by executing these artifacts after reducing them to workflows.

To implement the ideas behind the proposed architecture, we described an HTN ontology to define agent plans, developed a workflow component using Java, and focused on the reduction of agent plans to workflows. We used this planner architecture in industrial and academical projects. The last version of the SEAGENT can be downloaded[5] as an open source project.

SEAGENT planner has been designed with the idea that different plan definition languages other than HTN can also be reduced to the generic workflow model. In addition, business process definition languages such as BPEL, OWL-S can also be reduced to the generic workflow model. Moreover, these business process definition languages can be used in connection with different plan definition languages providing the interoperability of them. These features show the way of incorporating different languages into agent programming paradigms as well as offering a high degree of flexibility in developing agent systems.

[5]SEAGENT Semantic Web Enabled Framework, http://seagent.ege.edu.tr/

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovicand, and S. Weerawarana, "Business process execution language for web services v-1.1," W3C, Candidate Recommendation, 2003.

[2] WfMC, "Workflow management coalition workflow standard: Workflow process definition interface - xml process definition language (xpdl) (wfmc-tc-1025)," Workflow Management Coalition, Lighthouse Point (FL), Tech. Rep., 2002.

[3] K. Erol, J. Hendler, and D. S. Nau, "Semantics for hierarchical task-network planning," College Park, MD, USA, Tech. Rep., 1994.

[4] K. Sycara, M. Williamson, and K. Decker, "Unified information and control flow in hierarchical task networks," in *Working Notes of the AAAI-96 workshop 'Theories of Action, Planning, and Control'*, 1996.

[5] W. Sadiq and M. Orlowska, "Modeling and verification of workflow graphs," in *Technical Report No. 386, Department of Computer Science. The University of Queensland, Australia*, 1996.

[6] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau, "Htn planning for web service composition using shop2," *J. Web Sem.*, vol. 1, no. 4, pp. 377–396, 2004.

[7] G. Caire, D. Gotta, and M. Banzi, "Wade: a software platform to develop mission critical applications exploiting agents and workflows," in *AAMAS (Industry Track)*, 2008, pp. 29–36.

[8] A. P. F. Bellifemine and G. Rimassa, "JADE - a FIPA-compliant agent framework," in *Proceedings of the Practical Applications of Intelligent Agents*, 1999.

[9] E. E. Ekinci, A. M. Tiryaki, Ö. Gürcan, and O. Dikenelli, "A planner infrastructure for semantic web enabled agents," in *OTM Workshops*, 2007, pp. 95–104.

[10] O. Dikenelli, "Seagent mas platform development environment," in *AAMAS (Demos)*, 2008, pp. 1671–1672.

[11] J. R. Graham, K. Decker, and M. Mersic, "Decaf - a flexible multi agent system architecture." *Autonomous Agents and Multi-Agent Systems*, vol. 7, no. 1-2, pp. 7–27, 2003.

[12] K. P. Sycara, M. Paolucci, M. V. Velsen, and J. A. Giampapa, "The retsina mas infrastructure," *Autonomous Agents and Multi-Agent Systems*, vol. 7, no. 1-2, pp. 29–48, 2003.

[13] W. M. P. van der Aalst, "The application of petri nets to workflow management," *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[14] S. W. Sadiq, M. E. Orlowska, W. Sadiq, and C. Foulger, "Data flow and validation in workflow modelling," in *ADC*, 2004, pp. 207–214.

[15] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Inf. Syst.*, vol. 25, no. 2, pp. 117–134, 2000.

[16] J. Davis, W. Du, and M.-C. Shan, "Openpm: An enterprise process management system," *IEEE Data Eng. Bull.*, vol. 18, no. 1, pp. 27–32, 1995.

[17] W. Du, J. Davis, and M. C. Shan, "Flexible specification of workflow compensation scopes," in *GROUP '97: Proceedings of the international ACM SIGGROUP conference on Supporting group work*. New York, USA: ACM, 1997, pp. 309–316.

[18] B. K. W.M.P van der Aalst, A.H.M. ter Hofstede and A. Barros, "Workflow patterns," in *Distributed and Parallel Databases*, July 2003, pp. 5–51.