

A New HTN Planning Framework for Agents in Dynamic Environments

Hisashi Hayashi, Kenta Cho, and Akihiko Ohsuga

Knowledge Media Laboratory
Corporate Research and Development Center
TOSHIBA CORPORATION

1 Komukai-Toshiba-cho, Saiwai-ku, Kawasaki-shi, 212-8582, Japan
{hisashi3.hayashi, kenta.cho, akihiko.ohsuga}@toshiba.co.jp

Abstract. In a dynamic environment, even if an agent makes a plan to obtain a goal, the environment might change while the agent is executing the plan. In that case, the plan, which was initially valid when it was made, might later become invalid. Furthermore, in the process of replanning, it is necessary to take into account the side effects of actions already executed. To solve this problem, we have previously presented an agent life cycle that interleaves HTN planning, action execution, knowledge updates, and plan modification. In that agent life cycle, the plans are always kept valid according to the most recent knowledge and situation. However, it deals with only total-order plans. This paper extends the agent life cycle so that the agent can handle partial-order plans.

1 Introduction

As agents are working on the Internet, adaptation to dynamic environments is becoming an important subject. Because environments change, even if the agent obtained a goal in the past, it does not mean that the agent can obtain the same goal in the same way. Making inferences is an effective approach to adapt to such a dynamic environment. In fact, mobile agents of *Jini*[17] and *MiLog*[6] make inferences using Prolog. Mobile agents of *Plangent*[14] make a plan before acting.

If an agent makes a complete plan just before acting, the agent can adapt to the dynamic world to some extent. However, this is not enough: the agent might not be able to execute some actions as expected; the agent might get new information which affects the plan. In the former case, the agent needs to suspend the plan currently being executed and switch to an alternative plan. In the latter case, the agent needs to check the validity of the plan and modify it if necessary. Therefore, in order to adapt to the changing environment, it is necessary to check and modify plans continually and incrementally.

As pointed out in [4], much of the research into continual, distributed planning is built around the notion of abstract plan decomposition such as *hierarchical task network (HTN) planning*. HTN planners, such as *SHOP*[13], decompose a task to more primitive tasks. If an agent keeps an abstract plan, the agent can

successively decompose the abstract tasks in the plan as it gets new information from another agent, the user, or the environment.

Unlike standard partial-order planners, when planning, HTN planners do not always connect preconditions and postconditions (effects) of actions. Instead, such HTN planners decompose an abstract task into a more detailed plan. In this approach, plan decomposition is done by replacing the abstract task into the ready-made plan that is chosen from the plan library. Obviously, in this case, the agent can save the time to construct the ready-made plan. Note that this task decomposition is similar to the literal decomposition of Prolog. Prolog decomposes a literal using a clause that is selected from the Prolog program. The clause shows a plan (the body of the clause) for the abstract task (the head of the clause), and the Prolog program corresponds to the plan library.

Although the task (or literal) decomposition of HTN planners or Prolog is useful for agents in dynamic environments, we need to extend them so that they can incrementally and dynamically modify the plans. This is an important subject, which is recognized also in [5]. To deal with this problem, we previously designed an HTN planner [9] for speculative computation in multi-agent systems that interleaves planning, action execution, knowledge updates, and plan modifications. We also implemented the dynamic planning mobile agent system [10] on top of our *picoPlangent* mobile agent system. Based on Prolog's proof procedure, this planner makes (possibly more than one) plans, and incrementally modifies the plans. When one plan does not work or other plans become more attractive, the agent switches to another plan. The agent checks and modifies plans when executing an action or updating the program (Prolog's clauses). Plans are checked and modified after executing an action in a plan because this action might prevent the execution of the other alternative plans. Plans are also checked and modified after updating the program because some plans, which were initially valid when they were made, might become invalid and also it might be possible to make new valid plans after the program update.

Our previous HTN planner made it possible to guarantee the validity of plans with regard to the newest knowledge of the agent and its history of action execution. However, it can handle only *total-order* plans, which causes another problem: when switching from one plan to another, the agent sometimes cancels an action and re-executes the same action. For example, consider the following two total-order plans:

- [buy(a), buy(b), assemble(pc)]
- [buy(c), buy(a), assemble(pc)]

Informally, according to the first plan, the agent is expected to buy the part **a**, buy the part **b**, and then assemble **pc**. (See Figure 1.) The second plan is interpreted in the same way. Suppose that the agent has executed the action **buy(a)**. Our previous planner modifies the plans as follows:

- [buy(b), assemble(pc)]
- [return(a), buy(c), buy(a), assemble(pc)]

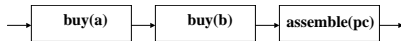


Fig. 1. A Total-Order Plan

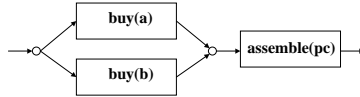


Fig. 2. A Partial-Order Plan

From the first plan, the executed action (`buy(a)`) has been removed. There is no problem with regard to the first plan. However, according to the second plan, the agent is expected to `return(a)` and later `buy(a)`. Although the second plan is correct, it is wasteful to redo what is undone. This problem is caused because the second total-order plan specifies the order of buying the parts `a` and `c` although the order is not a matter of concern. If we can use *partial-order* plans, we do not have to specify the order of `buy(a)` and `buy(c)`. In order to avoid this kind of unnecessary execution of canceling actions, we have extended our previous planner so that it can handle (restricted) partial-order plans. If we can use partial-order plans, we can express plans such as:

- $[\{\text{buy}(\mathbf{a}), \text{buy}(\mathbf{b})\}, \text{assemble}(\mathbf{pc})]$
- $[\{\text{buy}(\mathbf{c}), \text{buy}(\mathbf{a})\}, \text{assemble}(\mathbf{pc})]$

In the first plan, the execution order of `buy(a)` and `buy(b)`, which is written between “{” and ”}”, is not specified. (See Figure 2.) Similarly, the second plan does not specify the execution order of `buy(c)` and `buy(a)`. Therefore, after executing `buy(a)`, these plans can be modified as follows:

- $[\text{buy}(\mathbf{b}), \text{assemble}(\mathbf{pc})]$
- $[\text{buy}(\mathbf{c}), \text{assemble}(\mathbf{pc})]$

In summary, this paper presents a new agent life cycle that deals with (restricted) partial-order plans and that integrates the following:

- Decomposition of abstract tasks (literals);
- Action execution and plan modifications;
- Program updates and plan modifications.

This paper is organized as follows. In the next section, we define basic terminologies. Section 3 defines the procedure for task (literal) decomposition. This planning (task decomposition) procedure records extra information to prepare for future plan modifications. Section 4 shows how to modify the plans after action execution. Plans are modified considering the side effects of actions. Section 5 shows how to modify the plans after a program update: invalid plans are removed; new valid plans are added. Section 6 shows how to integrate task decomposition, action execution, program updates, and plan modifications.

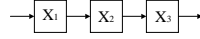


Fig. 3. Execution Order of a Plan (Type 1)

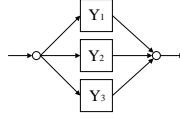


Fig. 4. Execution Order of a Plan (Type 2)

2 Terminology

This section defines basic terminologies including plans, clauses, dynamic planning framework, action data, and history of action execution. Like Prolog, (positive) literals will be defined using *constants*, *variables*, *functions*, and *predicates*.

Definition 1. A **complex term** is of the form: $F(T_1, \dots, T_n)$ where $n \geq 0$, F is an n -ary function, and each T_i ($1 \leq i \leq n$) is a term. A **term** is one of the following: a constant; a variable; a complex term. A **literal** is of the form: $P(T_1, \dots, T_n)$ where $n \geq 0$, P is an n -ary predicate, and each T_i ($1 \leq i \leq n$) is a term. When P is a 0-ary predicate, the literal $P()$ can be abbreviated to P .

Plans are defined using literals as follows.

Definition 2. A **Plan** is either **Type 1** of the form: $[X_1, \dots, X_n]$ or **Type 2** of the form: $\{Y_1, \dots, Y_n\}$ where $n \geq 0$, each X_i and each Y_i ($1 \leq i \leq n$) is a literal or a plan, which is called a **subplan**. The plan $[]$ and the plan $\{\}$ are called **empty plans**.

Intuitively, Figure 3 expresses the execution order of a plan (Type 1): $[X_1, X_2, X_3]$, and Figure 4 expresses the execution order of a plan (Type 2): $\{Y_1, Y_2, Y_3\}$. These execution orders will be reflected in the definition of *action consumability* of plans.

Example 1. Figure 5 expresses the execution order of a plan (Type 2): $\{[a1, a2], a3, [a4, a5]\}$ in which two subplans of Type 1 ($[a1, a2]$ and $[a4, a5]$) occur.

We define clauses using plans. Unlike Prolog’s clauses, the execution order of literals in a clause is partially specified.

Definition 3. A **clause** defining the literal LIT by the plan PLAN is of the form: $LIT \Leftarrow PLAN$. When PLAN is an empty plan, this clause is called a **fact** and can be abbreviated to LIT.

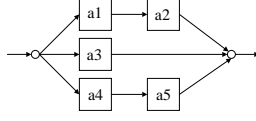


Fig. 5. Plan which has subplans

Intuitively, in the above definition, $LIT \Leftarrow PLAN$ means that LIT is satisfied when the execution of PLAN is completed.

Next, we define the *dynamic planning framework*, which will be used for our planning procedure.

Definition 4. A **dynamic planning framework** is of the form: $\langle A, D, P \rangle$ where A is a set of literals (called **actions**), D is a set of literals (called **dynamic literals**) that are not actions, P is a set (called **program**) of clauses that do not define any action. An instance of an action is also an action. An instance of a dynamic literal is also a dynamic literal.

In the above definition, actions are executed by an agent. The program will be used for planning. The definition clauses of dynamic literals in the program might be updated in the future by adding (or deleting) a clause to (respectively, from) the program.

Following the intuitive execution order of plans, we define action consumability of plans as follows. Informally, a plan can consume an action if the action is one of the next actions to be executed according to the plan.

Definition 5. In the dynamic planning framework, the action A is **consumable** for the plans of the form:

1. $[X_1, \dots, X_n]$ where
 - $n \geq 1$, and X_1 is either an action that is unifiable with A or a subplan such that A is consumable.
2. $\{Y_1, \dots, Y_n\}$ where
 - $n \geq 1$, and there exists Y_i ($1 \leq i \leq n$) such that Y_i is either an action that is unifiable with A or a subplan such that A is consumable.

Definition 6. In the dynamic planning framework, the action A is **inconsumable** for the plans of the form:

1. $[X_1, \dots, X_n]$ where
 - $n \geq 1$ and X_1 is either an action that is not unifiable with A or a subplan such that A is inconsumable.
2. $\{Y_1, \dots, Y_n\}$ where
 - $n \geq 0$ and each Y_i ($1 \leq i \leq n$) is either an action that is not unifiable with A or a subplan such that A is inconsumable.

Example 2. In a dynamic planning framework, suppose that a_1 , a_2 , a_3 , a_4 , and a_5 are actions. The plan $\{[a_1, a_2], a_3, [a_4, a_5]\}$ can consume a_1 , a_3 , and a_4 but cannot consume a_2 and a_5 . (See Figure 5.)

An action execution in a plan might affect the execution of other plans. The *action data* of an action, defined as follows, will be used to record the data regarding the side effects of the action.

Definition 7. *In the dynamic planning framework, the action data of the action A is one of the following forms:*

- $dataA(A, noSideEffect)$
- $dataA(A, undoAction(seq, A^-))$
- $dataA(A, undoAction(con, A^-))$
- $dataA(A, cannotUndo)$

Informally, in the above definition, each action data has the following meaning:

- A does not have side effects.
- A has side effects. A^- is the action that will undo the execution of A unless other actions with side effects are executed between the execution of A and A^- .
- A has side effects. A^- is the action that will undo the execution of A . We do not care when A^- undoes A .
- A has side effects that cannot be undone.

Finally, we define the *history of action execution*, which records the order of actions that have been executed recently. It also records the side effects of actions already executed. This information will be used for plan modification.

Definition 8. *A history of action execution is of the form: $[H_1, \dots, H_n]$ where $n \geq 0$, and each H_i ($1 \leq i \leq n$) is an action data.*

3 Decomposition of Abstract Tasks (Literals)

This section introduces a new procedure for planning via literal decomposition. In the dynamic planning framework, the program might be updated even after making plans. The planning procedure adds extra information to plans so that the plans can be modified when necessary. A *plan plus*, defined below, records such extra information (clauses and history of action execution) in association with a plan.

Definition 9. *In the dynamic planning framework, a **plan plus** is of the form: (PLAN, CSET, HISTORY) where PLAN is a plan, CSET is a set of clauses (called **prerequisite clauses** of PLAN) in the program, and HISTORY is history of action execution.*

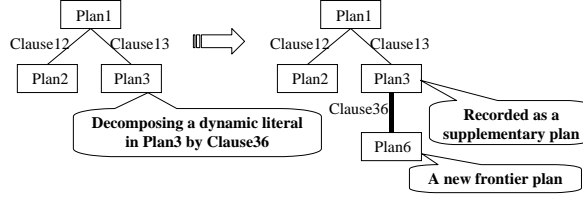


Fig. 6. Search Tree

Prerequisite clauses of a plan were used to derive the plan. Therefore, if a prerequisite clause of a plan becomes invalid, the plan becomes invalid. The history of action execution (See Definition 7) that is recorded in association with a plan records actions already executed, which have not been reflected in the plan yet.

Two kinds of plan pluses will be used by our planning procedure. The following *plan goal* records these two kinds of plan pluses.

Definition 10. *In the dynamic planning framework, a **plan goal** is of the form: $(\text{PLANS}^+, \text{SPP}^+)$ where PLANS^+ and SPP^+ are sets of plan pluses. Each plan plus in PLANS^+ is called a **frontier plan plus**. Each plan plus in SPP^+ is called a **supplementary plan plus**. The plan mentioned in a frontier plan plus is called a **frontier plan**. The plan mentioned in a supplementary plan plus is called a **supplementary plan**. For each supplementary plan, exactly one dynamic literal that occurs in the supplementary plan is **marked**.*

Frontier plans correspond to leaves in the search tree. In Figure 6, the left search tree means that Plan2 is derived from Plan1 using Clause12. Similarly, Plan3 is derived from Plan1 using Clause13. In this search tree, the frontier plans are Plan2 and Plan3. Our planning procedure will make plans by decomposing the selected literal in a frontier plan. If the selected literal is a dynamic literal, we record the frontier plan as a supplementary plan. For example, in Figure 6, when decomposing a dynamic literal in Plan3, Plan3 is recorded as a supplementary plan. Therefore, each supplementary plan has the selected dynamic literal, which is marked. Supplementary plans correspond to intermediate nodes (not leaves) from which another branch might be added. Note that a clause defining a dynamic literal might be added to the program later by a program update.

Like Prolog, a plan is resolved by a clause as follows.

Definition 11. *Let PLAN_v be a new variant¹ of PLAN. Suppose that the occurrence of the literal LIT_v in PLAN_v corresponds to the occurrence of the literal LIT in PLAN. Suppose that $\text{HEAD} \Leftarrow \text{BODY}$ is a new variant of the clause CL, and that LIT, LIT_v , and HEAD are unifiable.*

When LIT in PLAN is selected, the resolvent of PLAN by CL is made from PLAN_v by first unifying LIT_v and HEAD with the most general unifier θ , and then replacing the occurrence of $\theta(\text{LIT}_v)$ in $\theta(\text{PLAN}_v)$ with $\theta(\text{BODY})$.

¹ A new variant of X is made by replacing all the variables in X with new variables.

Example 3. When the literal $x2(b)$ in the plan $[x1(a), x2(b)]$ is selected, the resolvent of the plan $[x1(a), x2(b)]$ by the clause $x2(v) \Leftarrow \{y1(v), y2(v)\}$ is $[x1(a), \{y1(b), y2(b)\}]$.

Now, we define a *derivation* of plan goals. This derivation is our planning procedure. Not only does the derivation resolve plans, but it also records extra information which will be used for plan modification, when necessary.

Definition 12. *In the dynamic planning framework, a **derivation** from the plan goal $GOAL_1$ to the plan goal $GOAL_n$ ($n \geq 2$) is a sequence of plan goals: $GOAL_1, \dots, GOAL_n$ such that each $GOAL_{k+1}$ ($1 \leq k \leq n-1$) is derived from $GOAL_k$ ($= (PLANS^+, SPP^+)$) by one of the following **derivation rules**.*

- p1** *Select a plan plus $(PLAN, CSET, HISTORY)$ which belongs to $PLANS^+$. Select a literal L that occurs in $PLAN$ and that is not a dynamic literal. Suppose that C_1, \dots, C_s ($s \geq 0$) are all the clauses in the program such that each C_u ($0 \leq u \leq s$) defines a literal which is unifiable with the selected literal L . Suppose that $RPLAN_i$ ($1 \leq i \leq s$) is the resolvent of $PLAN$ by C_i . $GOAL_{k+1}$ is made from $GOAL_k$ by replacing the occurrence of the selected plan plus $(PLAN, CSET, HISTORY)$ with $(RPLAN_1, CSET, HISTORY), \dots, (RPLAN_s, CSET, HISTORY)$.*
- p2** *Select a plan plus $(PLAN, CSET, HISTORY)$ which belongs to $PLANS^+$. Select a literal L that occurs in $PLAN$ and that is a dynamic literal. Suppose that C_1, \dots, C_s ($s \geq 0$) are all the clauses in the program such that each C_u ($0 \leq u \leq s$) defines a literal which is unifiable with the selected literal L . Suppose that $RPLAN_i$ ($1 \leq i \leq s$) is the resolvent of $PLAN$ by C_i . $GOAL_{k+1}$ is made from $GOAL_k$ by replacing the occurrence of the selected plan plus $(PLAN, CSET, HISTORY)$ with $(RPLAN_1, \{C_1\} \cup CSET, HISTORY), \dots, (RPLAN_s, \{C_s\} \cup CSET, HISTORY)$, and adding the selected plan plus $(PLAN, CSET, HISTORY)$ to SSP^+ as a supplementary plan plus with the selected literal marked.*

Rule p1 decomposes a literal whose definition clause will not be updated. On the other hand, Rule p2 decomposes a dynamic literal whose definition clauses might be updated. When resolving a plan by a clause, Rule p2 records the clause in association with the resolved plan because if the clause becomes invalid, the resolved plan also becomes invalid. (See Figure 9.) Rule p2 also records the resolved plan as a supplementary plan because if another new clause which defines the selected literal becomes valid, it is possible to resolve the plan by the new valid clause.

Example 4. In the dynamic planning framework, suppose that $x2(v)$ is a dynamic literal. When resolving the plan $[x1(a), x2(b)]$, as shown in Example 3, using the clause $x2(v) \Leftarrow \{y1(v), y2(v)\}$, Rule p2 records this clause in association with the plan $[x1(a), \{y1(b), y2(b)\}]$. Also Rule p2 records $[x1(a), x2(b)]$ as a supplementary plan with the selected literal $x2(b)$ marked.

4 Action Execution and Plan Modification

When one plan does not work, we might want to suspend the plan execution and switch to another plan. However, some actions that have been executed following a plan might prevent another plan from functioning. This section defines how to modify plans after an action execution.

Definition 13. *In the dynamic planning framework, the plan modification rules after the execution of the action A are as follows where $PLAN$ is the plan to be modified by a plan modification rule, and only one plan modification rule is applied to $PLAN$ if more than one plan modification rule can be applied to $PLAN$.*

- *When A is consumable for $PLAN$:*
 - a1** *If $PLAN$ is of the form: $[A', X_1, \dots, X_n]$, $n \geq 0$, and A and A' are unifiable, then unify A and A' with the most general unifier θ and modify $PLAN$ to $\theta([X_1, \dots, X_n])$.*
 - a2** *If $PLAN$ is of the form: $[SUBPLAN, X_1, \dots, X_n]$, $n \geq 0$, and $SUBPLAN$ is a subplan, then modify $SUBPLAN$ following the plan modification rules after the execution of A .*
 - a3** *If $PLAN$ is of the form: $\{X_1, \dots, X_n\}$, $n \geq 1$, and X_i ($1 \leq i \leq n$) is unifiable with A , then unify X_i and A with the most general unifier θ , and remove $\theta(X_i)$ from $\theta(PLAN)$.*
 - a4** *If $PLAN$ is of the form: $\{X_1, \dots, X_n\}$, $n \geq 1$, and X_i ($1 \leq i \leq n$) is a subplan which can consume A , then modify X_i following the plan modification rules after the execution of A .*
- *When A is inconsumable for $PLAN$ and $DATA$ is the action data of A :*
 - b1** *If $DATA$ is of the form: $dataA(A, noSideEffect)$, then $PLAN$ is not modified.*
 - b2** *If $DATA$ is of the form: $dataA(A, undoAction(seq, A^-))$, then modify $PLAN$ to $[A^-, PLAN]$.*
 - b3** *If $DATA$ is of the form: $dataA(A, undoAction(con, A^-))$, then modify $PLAN$ to $\{A^-, PLAN\}$.*
 - b4** *If $DATA$ is of the form: $dataA(A, cannotUndo)$, then delete $PLAN$.*

When $PLAN$ can consume A , Rules a1-a4 removes A from the plan because A has been executed. When $PLAN$ cannot consume A , one of the rules b1-b4 is applied to the plan. If A does not have side effects (b1), we do not have to care. If A has side effects (b2, b3, b4), it is necessary to undo A . As shown in Figure 7, some A have to be undone before $PLAN$ is executed (b2). As shown in Figure 8, some A can be undone anytime (b3). If A cannot be undone (b4), it is impossible to use $PLAN$.

Example 5. In the dynamic planning framework, let **a1**, **a2**, **a3**, **a4**, **a5**, and **c5** be actions. Suppose that $dataA(a5, undoAction(seq, c5))$ is the action data of **a5**. After executing **a5**, the action modification rule b2 will modify the plan $\{[a1, a2], a3, [a4, a5]\}$, which was shown in Figure 5, to $[c5, \{[a1, a2], a3, [a4, a5]\}]$.

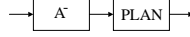


Fig. 7. Sequentially Undoing Action

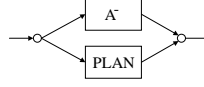


Fig. 8. Concurrently Undoing Action

The plan modification rules after the execution of the action A can be applied to a plan only if it is possible to judge whether the plan can consume A or not. For example, if the literal `lit1` is not an action, it is impossible to apply the plan modification rules to the plan `[lit1, lit2]`. However, when the plan is resolved further and the literal `lit1` is decomposed into an action, we can apply the plan modification rules to the resolved plan. In order to apply plan modification rules later to this kind of plan, we record the history of action execution in association with the plan.

Definition 14. *In the dynamic planning framework, the plan plus modification rules after the execution of the action A are as follows where $(PLAN, CSET, [H_1, \dots, H_n])$ ($n \geq 0$) is the plan plus to be modified by a plan plus modification rule.*

- c1** *If $n = 0$ and A is either consumable or inconsumable for $PLAN$, then modify $PLAN$ by applying a plan modification rule (defined in Definition 13) to $PLAN$.*
- c2** *If $n \geq 1$ or A is neither consumable nor inconsumable for $PLAN$, update the history of action execution from $[H_1, \dots, H_n]$ to $[H_1, \dots, H_n, H_{n+1}]$ where H_{n+1} is the action data of A .*

Finally, we define how to modify a plan goal after an action execution.

Definition 15. *In the dynamic planning framework, the plan goal modification rule after the execution of the action A is as follows where $PGOAL$ is the action to be modified by the plan goal modification rule:*

- d1** *Modify each plan plus that occurs in $PGOAL$ by the plan plus modification rule (defined in Definition 14) after the execution of A .*

Even if an action is neither consumable nor inconsumable for a plan, after further resolving the plan, the action might become consumable or inconsumable for the resolved plan. In that case, we modify the resolved plan using the information that is recorded in the history of action execution.

Definition 16. *In the dynamic planning framework, the plan plus modification rule after resolution is as follows where $(PLAN, CSET, [H_1, \dots, H_n])$ ($n \geq 0$) is the plan plus to be modified by a plan plus modification rule after resolution.*

- e1 If $n = 0$, then the plan plus is not modified.
- e2 If $n \geq 1$, H_1 is the action data of A , and A is neither consumable nor inconsumable, then the plan plus is not modified.
- e3 If $n \geq 1$, H_1 is the action data of A , and A is either consumable or inconsumable, then the plan plus is modified to: $(\text{NEWPLAN}, \text{CSET}, [H_2, \dots, H_n])$ where NEWPLAN is made from PLAN by applying a plan modification rule after the execution of A (Defined in Definition 13). If possible, further modify the plan plus in the same way.

Definition 17. In the dynamic planning framework, the plan goal modification rule after resolution is as follows where PGOAL is the plan goal to be modified by the plan goal modification rule:

- f Modify each frontier plan plus that occurs in PGOAL by the plan plus modification rule after the execution of A (defined in Definition 16).

5 Program Updates and Plan Modification

This section defines how to modify plans when the program is updated. A program update is done by clause addition or clause deletion, which affects the derivation that is based on the program: the addition (or deletion) of a clause adds (respectively, cuts) branches to (respectively, from) the search tree of the derivation. (See Figure 9.) When adding a clause to the program, we make new valid plans, if possible, using the added clause. When deleting a clause from the program, we delete the invalid plans that are derived using the deleted clause.

Definition 18. In the dynamic planning framework, the **plan goal modification rules after a program update** are as follows where $(\text{PLANS}^+, \text{SSP}^+)$ is the plan goal to be modified by a plan goal modification rule:

- When adding a clause C , which defines the literal LIT , to the program:
 - g1 Suppose $(\text{PLAN}_1, \text{CSET}_1, \text{HISTORY}_1), \dots, (\text{PLAN}_n, \text{CSET}_n, \text{HISTORY}_n)$ are all the supplementary plan pluses in SSP^+ such that $n \geq 0$ and the marked literal in each supplementary plan PLAN_k ($1 \leq k \leq n$) is unifiable with LIT . Let RPLAN_i be the resolvent of PLAN_i ($1 \leq i \leq n$) by C . The plan goal $(\text{PLANS}^+, \text{SSP}^+)$ is modified by adding each $(\text{RPLAN}_i, \{C\} \cup \text{CSET}_i, \text{HISTORY}_i)$ ($1 \leq i \leq n$) to PLANS^+ .
- When deleting the clause C from the program:
 - g2 The plan goal $(\text{PLANS}^+, \text{SSP}^+)$ is modified by deleting each plan plus that records C as a prerequisite clause.

The clause that might be added to the program defines a dynamic literal. In Figure 9, when resolving Plan 3 by decomposing a dynamic literal L , following the derivation rule p2, we record Plan 3 as a supplementary plan. Therefore, even if Clause 37 that defines L is added to the program after the resolution, we

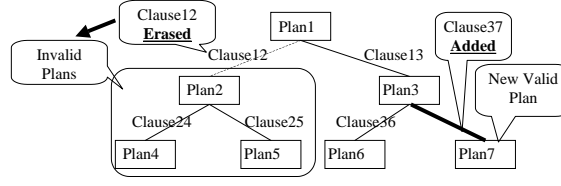


Fig. 9. A search tree

can still make Plan 7 resolving Plan 3 by Clause 37. Note that this resolution corresponds to the addition of a new branch to the search tree.

On the other hand, if a clause is deleted from the program, the plans that were derived using the deleted clause become invalid. In Figure 9, Plan 2, Plan 4, and Plan 5 depend on Clause 12. If Clause 12 defines a dynamic literal, the derivation rule p2 records Clause 12, as a prerequisite clause, in association with Plan 2. In this case, Clause 12 is recorded also in association with Plan 4 and Plan 5 because these plans are made by resolving Plan 2. Therefore, when deleting Clause 12 from the program, we can delete Plan 2, Plan 4, and Plan 5, which are the only plans that have become invalid. Note that this plan deletion corresponds to the deletion of a branch from the search tree.

6 Agent Life Cycle

This section defines an agent life cycle that integrates planning, action execution, program updates, and plan modifications. Given the *initial goal* (a literal), the agent makes plans to obtain the initial goal, selects a plan, and starts the plan execution. After executing an action or updating the program, the agent incrementally modifies its plans. When one of the plans becomes empty and its history of action execution becomes empty, the agent finishes its life cycle as the initial goal has been satisfied.

Definition 19. *In the dynamic planning framework, given a literal L as the initial goal, the agent life cycle is defined as follows:*

1. Let $\{L\}$ be the **initial plan** IPLAN. Let $(\text{IPLAN}, \{\}, \{\})$ be the **initial plan plus** IPLAN⁺. Let $(\{\text{IPLAN}^+\}, \{\})$ be the **initial plan goal** IPGOAL. Set the **current plan goal** to IPGOAL.
2. Repeat the following procedure until one of the frontier plans in the current plan goal becomes an empty plan and its history of action execution becomes empty:
 - (a) (Program Updates) Repeat the following procedure if necessary:
 - i. Add (or delete) a clause² C that defines a dynamic literal to (respectively, from) the current program.

² We assume that the agent has the module that decides how to update the program. This module chooses the clause to add or delete at Step 2(a)i.

- ii. After the program update, apply a plan goal modification rule, which is defined in Definition 18, to the current plan goal.
- (b) (Action Execution) Repeat the following procedure if necessary:
 - i. If possible, select a frontier plan which is mentioned in the current plan goal, select an action **A** which is consumable for the selected plan, and execute **A**.
 - ii. If the execution of **A** is successful, apply a plan goal modification rule, which is defined in Definition 15, to the current plan goal.
- (c) (Plan Decomposition) If possible, make a derivation from the current plan goal to another plan goal **PGOAL**, and update the current plan goal to **PGOAL**.
- (d) (Plan Modification Using History of Action Execution) Apply a plan goal modification rule, which is defined in Definition 17, to the current plan goal.
- (e) (Simplification of Plans) Repeat the following procedure, if possible:
 - i. Delete empty subplans³ of a plan that occurs in the current plan goal.
 - ii. If a subplan **SPLAN** of a plan that occurs in the current plan goal is of the form **{E}** or of the form **[E]**, where **E** is a literal or a subplan of **SPLAN**, then replace **SPLAN** with **E**.

Example 6. In the dynamic planning framework, suppose that **good(X)** is a dynamic literal, that **buy(X)** and **assemble(X)** are actions, and that the program is as follows:

```

make(X) ← [getPartsFor(X), assemble(X)].
getPartsFor(X) ← {parts(X, Y), get(Y)}.
get([]).
get([H|T]) ← {good(H), buy(H), get(T)}.
parts(pc, [a, b]) . parts(pc, [b, c]) . parts(pc, [c, a]) .
good(a) . good(b) . good(c).

```

Note that **good(a)**, **good(b)**, and **good(c)** are speculatively assumed to be true. (We do not know if those parts are good until we actually buy them.) Given the initial goal **make(pc)**, the agent makes the following three plans:

```

[ {buy(a), buy(b)}, assemble(pc) ]
[ {buy(b), buy(c)}, assemble(pc) ]
[ {buy(c), buy(a)}, assemble(pc) ]

```

According to the above plans, the agent is expected to buy some parts and assemble a PC. Note that the execution order of assembling the parts is not specified.

Now, suppose that the agent selects the action **buy(a)** in the first plan. Suppose that **dataA(buy(X), undoAction(con, return(X)))** is the action data of **buy(X)**. After executing **buy(a)**, the plans are modified as follows:

³ An empty plan is not deleted if it is not a subplan.

```
[buy(b), assemble(pc)]
{return(a), [{buy(b), buy(c)}, assemble(pc)]}
[buy(c), assemble(pc)]
```

From the first and third plan, the executed action `buy(a)` has been removed. In the second plan, `return(a)`, which will undo `buy(a)`, has been inserted. Note that the execution order of `return(a)` and the rest of the plan is not specified.

At this point, suppose that the agent has got the information that the part “a” is broken and erased `good(a)` from the program. Because the agent records `good(a)`, as a prerequisite clause, in association with the first plan and the third plan, the agent erases these two plans. As a result, only the valid second plan remains:

```
{return(a), [{buy(b), buy(c)}, assemble(pc)]}
```

Suppose that the agent has executed `buy(b)`. The agent erases `buy(b)` from the above plan.

```
{return(a), [buy(c), assemble(pc)]}
```

Suppose that the agent has found that the part “a” is not broken and added `good(a)` again to the program. The agent makes again the two plans that were erased when `good(a)` was deleted from the program.

```
{return(a), [buy(c), assemble(pc)]}
[assemble(pc)]
{return(b), [buy(c), assemble(pc)]}
```

Note that the execution of `buy(b)` has been reflected in the two revived plans: `[assemble(pc)]` and `{return(b), [buy(c), assemble(pc)]}`. This is because the supplementary plan from which `[assemble(pc)]` and `{return(b), [buy(c), assemble(pc)]}` are derived records the history of action execution.

7 Related Work

Our plan modification method after program updates is based on the proof procedure called *Dynamic SLDNF(DSLDNF)* [7, 8], which incrementally updates the program and modifies the computation. There is a proof procedure [16] of logic programming which is closely related to DSLDNF. When deleting an assumption (an abducible), this proof procedure modifies the computation in the same way as DSLDNF. Another proof procedure [11] of logic programming modifies the computation after observation. Once a fact has been observed, however, this proof procedure cannot delete observed facts from the knowledge base. (This is not a defect for a system which never deletes data.) In [12], the agent can assimilate clauses of generalized logic programming, and a special semantics is used to interpret the possibly contradictory knowledge.

Transaction logic programming (TLP)[2] is related to our plan modification method after action execution in Section 4. Like Prolog, the procedure in [2] executes literals in the body of a clause from left to right. When backtracking, it undoes some actions. There is also a concurrent version [1] of TLP. Our procedure is different from TLP because our procedure can modify the plans without backtracking. The importance of canceling actions is recognized also in the area of web services. For example, according to the specification of *BPEL4WS*[3], if an activity in a scope is not executed as expected, the activities already executed in the scope will be undone.

The standard replanning method of partial-order planning is explained in such a standard AI textbook as [15]. In the standard partial-order planning algorithm, the plan is made by connecting “preconditions” and “postconditions (effects)” of actions, which is different from our planning procedure. Our planner makes plans by decomposing literals, which is closer to HTN planners such as SHOP[13]. As mentioned in [5], replanning in dynamic environments is the important future work for SHOP.

8 Conclusion and Future Work

We have shown how to integrate task decomposition of HTN planning, action execution, program updates, and plan modifications. When updating the program or executing an action, our procedure incrementally modifies the plans. Although our procedure does not use the knowledge describing the effects of actions on the state of the world expressed by fluents, it takes into account the side effects of actions when modifying plans after action execution.

Compared with our previous work, we have improved our planning procedure using (restricted) partial-order plans, by which we can further avoid unnecessary cancellation of already executed actions when switching from one plan to another. In addition, we introduced a new type of undoing actions that can be executed anytime, which also contributes to the avoidance of action cancellation.

As mentioned in the previous section, the plan modification method after a program update is based on the proof procedure of DSLDNF. When updating the program, invalid plans are removed and new valid plans are added. In [7, 8], it was shown that DSLDNF generally replans faster than SLDNF. This is because DSLDNF reuses already derived plans while SLDNF has to recompute from scratch.

Incremental plan modification of our planner is also useful when plans are partially decomposed or refined using the information from another agent or the user. If replanned from scratch, the agent will forget all those refined plans.

As we did for our previous planning agent system [10, 9], we would like to introduce an independent semantics as soon as possible. We are also interested in the research into the selection strategy of actions and plans. Another subject for future work is handling the interference between subplans. Finally, the most important subject for future work for us is application development although we have tested our new planning algorithm and confirmed it works.

References

1. A. J. Bonner and M. Kifer. Transaction Logic Programming. International Conference on Logic Programming, pp. 257-279, 1993.
2. A. J. Bonner. Workflow, Transactions and Datalog. ACM Symposium on Principles of Database Systems, pp. 294-305, 1999.
3. BPEL4WS v1.1 Specification, 2003.
4. M. E. desJardins, E. H. Durfee, C. L. Ortiz, Jr., and M. J. Wolverton. A Survey of Research in Distributed, Continual Planning. *AI Magazine*, pp. 12-22, Winter 1999.
5. J. Dix, H. Munoz-Avila, and D. Nau. IMPACTing SHOP: Putting an AI Planner into a Multi-Agent Environment. *Annals of Mathematics and AI*, 4(37):381-407, 2003.
6. N. Fukuta, T. Ito, and T. Shintani. MiLog: A Mobile Agent Framework for Implementing Intelligent Information Agents with Logic Programming. Pacific Rim International Workshop on Intelligent Information Agents, 2000.
7. H. Hayashi. Replanning in Robotics by Dynamic SLDNF. IJCAI Workshop "Scheduling and Planning Meet Real-Time Monitoring in a Dynamic and Uncertain World", 1999.
8. H. Hayashi. Computing with Changing Logic Programs. PhD Thesis, Imperial College of Science, Technology and Medicine, University of London, 2001.
9. H. Hayashi, K. Cho, A. Ohsuga. Speculative Computation and Action Execution in Multi-Agent Systems. ICLP Workshop on Computational Logic and Multi-Agent Systems (CLIMA), *Electronic Notes in Theoretical Computer Science* 70(5), <http://www.elsevier.nl/locate/entcs/volume70.html>, 2002.
10. H. Hayashi, K. Cho, and A. Ohsuga. Mobile Agents and Logic Programming. IEEE International Conference on Mobile Agents, pp. 32-46, 2002.
11. R. A. Kowalski and F. Sadri. From Logic Programming to Multi-Agent Systems. *Annals of Mathematics and Artificial Intelligence*, 1999.
12. J. A. Leite, J. J. Alferes, and L. M. Pereira. MINERVA-A Dynamic Logic Programming Agent Architecture. International Workshop on Agent Theories, Architectures, and Languages, 2001.
13. D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. International Joint Conference on Artificial Intelligence, pp. 968-975, 1999.
14. A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden. PLANGENT: An Approach to Making Mobile Agents Intelligent. *IEEE Internet Computing*, 1(4):50-57, 1997.
15. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.
16. K. Satoh. Speculative Computation and Abduction for an Autonomous Agent, International Workshop on Non-Monotonic Reasoning, pp. 191-199, 2002.
17. P. Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog, International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology, 1999.