Interleaving Temporal Planning and Execution: IxTeT-eXeC*

Solange Lemai and **Félix Ingrand**†
LAAS/CNRS,
7 Avenue du Colonel Roche, F-31077 Toulouse Cedex 04, France
{*slemai,felix*}*@laas.fr*

Abstract

Execution control of plans is a very active domain of research, but remains a major challenge when performed on board real autonomous systems such as robots or satellites. In such a context, where execution concurrency, resources contention and environment dynamic characterize the domain, the use of a temporal planner and a temporal execution control system is desirable. This paper presents IxTeT-eXeC, a recent extension of the temporal planner IxTeT which allows execution control, plan repair, and replanning when necessary. This paper is a short version of [Lemai & Ingrand 2003].

Introduction

IxTeT-eXeC enables the system to execute and monitor the execution of a temporal flexible plan. It takes into account runtime failures and timeouts from the underlying functional components and incorporates those failures in the plan. If some flexibility was left for the failed action, it may try another way to achieve it, otherwise it tries to repair the plan to still achieve the goal, and if this fails too, it replans the whole plan. Figure 1 presents how IxTeT-eXeC is integrated in the overall LAAS architecture, and how it relates to the procedural executive which relays the actions to the functional level and passes back the reports of success or failure of those same actions.

The paper is organized as follows. The first section describes the IxTeT-eXeC component and details the methods and algorithms used to perform temporal plan execution monitoring, plan repair and replanning. We illustrate the current state of implementation with an example, and conclude with a number of extensions currently being implemented.

IxTeT-eXeC

The IxTeT system [Ghallab & Laruelle 1994] is a lifted partial-order temporal planner based on CSPs. The temporal representation describes the world as a set of multi-valued functions of time, called $attributes$, and resources over which borrowing, consumption or production can be specified. The planner deals with a set of deterministic operators called *tasks*. A *task* is a temporal structure composed of a set of *events* describing the change of the world induced by the task, a set of *hold* assertions on attributes to express required conditions or the protection of some fact between two events (causal link), a set of resource usages, and a set of temporal and binding constraints on the different timepoints and variables of the task. The planning system IxTeT presents interesting properties in the context of plan execution and plan modification: it elaborates very flexible plans, partially ordered and partially instantiated and performs the search in the plan space. The purpose of IxTeT-eXeC is to extend IxTeT to interleave more closely planning and temporal execution, especially to: regularly update the plan under execution, reactively replan in case of failure, incrementally replan upon arrival of new goals.

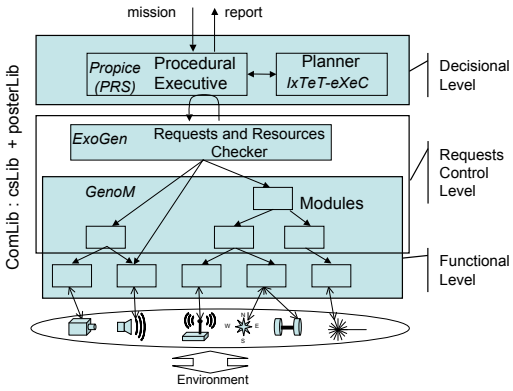

Figure 1: The LAAS Architecture.

The key component in IxTeT-eXeC is a temporal executive which interacts with the planning system. The general schema of execution is the following. First, given a description of the task operators and of an initial plan containing the initial situation and the goals, a complete plan is elaborated. This plan is then executed. At each step of the execution, the temporal executive selects the appropriate timepoints from the temporal network, sends the corresponding commands to the procedural executive for task expansion and integrates

*Part of this work was funded by a contract with CNES and ASTRIUM.

†This author is currently on sabbatical at NASA Ames Research Center, Moffett Field, CA, USA.

Execution Cycle
$ExecutedPlan$: plan currently under execution
$ExecutableTPs$: set of executable timepoints
t_{exec}: execution time of the next executable timepoint
$ExecTPs$: set of timepoints to execute during the cycle

1.cycle forever
2. wake up if (current_time $\leq t_{exec}$) or ($replan$)
   or ($MsgQueue$ not empty)
3. $cycle_start_time \leftarrow$current_time
4. $cycle\_end\_time \leftarrow cycle\_start\_time + timestep$
5. Sense()
6. PlanRepair()
7. Act()
8 get next t_{exec}
9. add executable TPs occuring at t_{exec} to $ExecTPs$
10.end cycle

Sense()
1.if ($MsgQueue$ not empty)
2. for each Msg
3. if ($report$ is nominal)
4. set_execution_time(cycle_start_time)
5. forget_the_ past()
6. if ($report$ is failed)
// partial invalidation of ExecutedPlan
7. if (cycle_start_time $\geq$ timepoint_lower_bound)
8. set_execution_time(cycle_start_time)
9. else
10. create_new_timepoint()
11. insert_new_state()
12. remove_cls_on_common_attributes()
13. forget_the_ past()
14. $replan \leftarrow$true
15. update $ExecutableTPs$, $ExecTPs$
16. $NewSearchTree \leftarrow$true

PlanRepair()
1.if ($replan$)
2. if ($NewSearchTree$)
3. set_searchtree_root($ExecutedPlan$)
4. $NewSearchTree \leftarrow$ false
5. get $limit\_time$
6. while (!$solution\_found$) and
(current_time $\leq limit\_time$)
7. $solution\_found$
$\leftarrow$plan_one_step($cycle\_end\_time$)
8. if ($solution\_found$)
9. $ExecutedPlan \leftarrow$ Solution Plan
10. $replan \leftarrow$ false
11. else
12. $ExecutedPlan \leftarrow$ get_best_partial_plan()
13. update $ExecutableTPs$, $ExecTPs$

the reports sent in return. In case of failure, the temporal executive invalidates the part of the plan concerned by the failure. Then, taking advantage of the temporal flexibility of the plan and using IxTeT procedures, it tries to repair the plan while continuing the execution of its valid part. If this plan repair fails, the temporal executive aborts the execution, abandons the current plan and restarts a complete planning process from the new situation and the not yet achieved goals.

Temporal execution

The temporal executive controls the temporal network of the plan (a *minimal STN*, [Dechter, Meiri, & Pearl 1989]) to decide the execution of tasks and maps the abstract timepoints to their real execution time.

The timepoints of an IxTeT plan correspond to different types of event: start or end of a task, some contingent external event (as for instance the timepoints defining a visibility window for a space application), or some internal event of a task (used to represent the variation of a resource profile, or some more complex dependency between tasks ...). The current implementation only takes into account start and end timepoints.

Furthermore, three types of tasks are considered. *Non preemptive* tasks cannot be terminated by the controller and the end timepoint is uncontrollable. *Early preemptive* and *late preemptive* tasks can be terminated by the controller, as soon as possible in the first case, as late as possible otherwise. Note that IxTeT is not able to handle non controllable temporal variables and contingent durations can be squeezed during propagation. IxTeT-eXeC can only detect when an uncontrollable timepoint times out. Further work needs to be done to make the time-map manager verify the Dynamic Controllability property described in [Morris, Muscettola, & Vidal 2001].

The algorithms **Execution Cycle**, **Sense()**, **PlanRepair()** and **Act()** present how IxTeT-eXeC is implemented. After the elaboration of a complete plan by IxTeT, its execution is started. Each time the executive needs to do something, i.e. a message has been received, or it is time to execute some timepoint, or some plan repair process is in progress, it wakes up and follows the execution cycle described above.

In these algorithms, $ExecutedPlan$ refers to the plan being executed. At the beginning, it corresponds to the flexible plan resulting from the initial complete planning process. A timepoint of its temporal network is *executable* if all timepoints that must directly precede it have already been executed. The temporal executive determines what should be the next timepoint to execute and its execution time (t_{exec}). In fact, several timepoints may have to be executed during one cycle. The set of these timepoints, $ExecTPs$, is initialized with the set of timepoints occuring at t_{exec} and updated during the cycle with the new executable timepoints which have to be executed before the end of the cycle. The execution time of a timepoint depends on its type. It corresponds to the lower bound for start timepoints and end timepoints of early preemptive tasks; and to the upper bound for end timepoints of late preemptive and non preemptive tasks. In the last case, this "execution time" only corresponds to a deadline used to detect possible timeouts.

Two types of commands are sent to the procedural executive: $(START\ TaskId\ parameters)$ or $(END\ TaskId)$

```
Act()
1.while (ExecTPs not empty) and (!end_cycle)
  and (!timeout)
2.  (ExecTP,exec_time) ← get_first_TP(ExecTPs)
3.  ExecuteNow ←true
    //ExecTPs can contain timepoints to execute
    //in another cycle (if wake up for replan or Msg)
4.  if(exec_time > cycle_end_time)
5.    end_cycle ←true
6.  else
7.    if (replan)
8.      if (ExecTP is start TP)
9.        flaw ←check_starting_task()
10.       if (flaw)
11.         ExecuteNow ←false
12.         if (ExecTP_ub > cycle_end_time)
13.           add ExecTP to WaitingExecTPs
14.           suppress ExecTP from ExecTPs
15.         else timeout ←true
16.   if (ExecuteNow)
17.     if (exec_time ≤ cycle_start_time)
18.       if (ExecTP_ub ≥ cycle_start_time)
19.         exec_time ← cycle_start_time
20.       else timeout ←true
21.     if (ExecTP not controllable and not received)
22.       timeout ←true
23.     else
24.       if (ExecTP is start TP)
25.         set_execution_time(exec_time)
26.         forget_the_ past()
27.         NewSearchTree ←true
28.       send_command()
29.       update ExecutableTPs, ExecTPs
30.add WaitingExecTPs to ExecTPs
```

(if the task is preemptive). A task is fully instantiated just before starting its execution. A report is sent back by the procedural executive each time a task is completed, which is mapped into the end timepoint of the task. Note that the real execution time assigned to an end timepoint is the time at which the report message has been received. More precisely a completion report contains the following information: a completion status (nominal or failed), and in case of failure, the actual state. This state is described as the set of new values for the attributes of the task.

The *sense* part of the cycle integrates the messages from the procedural executive. In the nominal case, it amounts to assigning the current time to the end timepoint of the task and propagating this value in $ExecutedPlan$ (*set_execution_time()*). New executable timepoints may appear, and $ExecTPs$ is updated. For instance, $ExecTPs$ may now contain the start timepoints of parallel tasks immediately following the completed task, that will be executed in the same cycle. The failed case is detailed in the next subsection.

The *act* part of the cycle "executes" the timepoints in $ExecTPs$ according to their precedence constraints. *get_first_TP()* (Algo: **Act()**, line 2) determines which timepoint to handle next and its execution time according to the temporal network (lower/upper-bound). Line 4 checks if the timepoint has to be taken into account during the current cycle. If not, no other timepoint is due during this cycle. Otherwise, the "execution" of the timepoint depends on its type. For a start timepoint: its execution time is assigned the value determined lines 17-20 and propagated (line 25), the corresponding command is sent. For the end timepoint of a preemptive task: the command is sent, but the execution time is set only once the report message is received in the *sense* part of the next cycle. Finally, a timeout is detected if a non preemptive task is not terminated yet, but should be (line 21). Each time a new timepoint is instantiated, $ExecutableTPs$ and $ExecTPs$ are updated (line 29).

The uncertainty on the duration of the execution cycle has some consequences on the exact execution time of start or end of tasks. The *timestep* (Algo: **Execution Cycle**, line 4) is an estimation of the maximal duration of the cycle. It is defined by the user and may vary with the application. The model description and the planning process are independent of the timestep. But the user has to be aware that two timepoints that have to be executed within an interval less than one timestep, will be executed during the same cycle according to their precedence constraints. Note that the cycle can possibly take less time, and then the executive can react to messages more quickly.

Finally, the temporal execution of a plan can lead to various needs for replan: (1) uncontrollable and controllable timepoints time out, (2) excessive use or insufficient production of resource, (3) new goals to insert, (4) failed tasks. The adopted strategy consists of two steps: repairing the plan (cases 2, 3 and 4) and executing its valid part while there remains some temporal flexibility; if this fails, aborting the execution and elaborating a new plan. The next subsection details the plan repair process.

Plan Repair

In most cases, failed tasks have not produced the effects initially expected in the plan. The plan repair consists of two steps. First, invalidate the part of the plan depending on these effects. This process includes removing the causal links supplied by the failed task, thus revealing new open conditions in the future. For the moment, the tasks present in the plan are not removed, to limit the amount of decisions. The second step tries to recover the lost properties of the plan by adding new tasks and resolving conflicts.

Partial invalidation of the plan Upon reception of a failure message, two situations may arise. If the reception time is consistent with the bounds of the end timepoint of the task, the task is considered to be finished and its end timepoint is instantiated (Algo: **Sense**, line 8). But the task can fail at any moment and before the minimal expected end time of the task. In that case, a new failure timepoint F is created (Algo: **Sense**, line 10) and set to the current time. It corresponds to the new end timepoint. The other timepoints of the task occuring after F are considered to be failed. Their temporal constraints are relaxed and the temporal propositions ($hold$, $event$, ...) are updated (eventually removed).

Precedence constraints are also added between F and the executable timepoints of the plan.

Next, the new state is inserted (Algo: **Sense**, line 11). The new state is formalized as a set of $events$ on the attributes of the task, occuring at the end timepoint (or at F) and setting the value of the attributes to the new values given by the procedural executive. Such an event may or may not be inserted in $ExecutedPlan$. If the plan does not contain any conflicting proposition with the event, it is inserted. If the new value is in conflict with propositions of another running task, it is not inserted. Indeed, we consider, that unless the other task is reported failed, its execution is nominal. Finally, if the new value is in conflict with some propositions of the failed task or some causal links, the event is inserted and the conflicting propositions and causal links are removed. Note that the breaking of causal links does not call the temporal constraints between tasks into question.

At this point, the plan may contain open conditions to reestablish. The repair may require the insertion of new tasks. To allow a task insertion within the current order of tasks, we need to break additional causal links (Algo: **Sense**, line 12). We adapted the work presented in [Gaborit 1996] to determine which causal links to remove. A plan repair is then attempted.

Interleave plan repair and execution The plan repair is similar to the IxTeT plan search process in the plan space. The root of the search tree is the partially invalidated plan $ExecutedPlan$. The search tree is developed according to an Ordered Depth First Search strategy.

Plan repair is distributed, if necessary, on several cycles to allow the concurrent execution of the valid part. During the *plan repair* part of the cycle, planning is done one step at a time until a solution is found or a deadline is reached. This deadline ($limit\_time$ on line 5, Algo: **PlanRepair**) corresponds to the share of the timestep allocated to the *sense* and *plan* parts. This parameter is defined by the user. This planning distribution raises two important problems:

1. On which plan relies the execution in the *act* part, especially if no solution has been found? This plan has to satisfy the condition: *The currently running tasks are fully supported in ExecutedPlan*. The plan does not contain any flaw in relation to these tasks. At each planning step, the node is labeled if the corresponding partial plan satisfies the condition. At the end of *PlanRepair* (line 12, Algo: **PlanRepair**) and if the current plan is not acceptable, the last labeled node is chosen and the corresponding plan becomes $ExecutedPlan$.

2. On which plan and which search tree relies the planning process in the next cycle? If no decision has been made meanwhile (no timepoint instantiation, no message reception), the search tree can be kept as is and further developed during the next *RepairPlan* part. It is even possible to backtrack on decisions made in previous cycles. However, if $ExecutedPlan$ has been modified, a new search tree is mandatory. Its root node is the new $ExecutedPlan$. The planning decisions made in the previous cycles are now fixed, no backtrack is possible.

Some precautions must be taken to prevent from planning in the past. Each new timepoint inserted during the planning process is constrained to occur after $cycle\_end\_time$. And, to prevent the planner from looking for threats or establishing events in the past, a $forget\_the\_past()$ function is applied at each timepoint instantiation. So that the sets used for the flaw analysis contain, for each instantiated attribute, only the last event and the assertions occuring after it.

The execution of a partially invalid plan requires to check, before starting a new task, that it is fully supported in $ExecutedPlan$ (Algo: **Act**, line 9). If not, and if the time upper bound of the start timepoint has not been reached, its execution is postponed (Algo: **Act**, line 13). In case of timeout, the execution is failed and a complete replanning process is necessary.

This plan repair process is not guaranteed to find a valid plan everytime (backtrack nodes frozen by execution or temporal constraints too tight to add new tasks ...), but can avoid to abort execution and completely replan at each failure. By invalidating only a part of the plan, the amount of decisions is rather limited and a repaired plan may be found in a few cycles. Plan repair is especially efficient and useful for not temporally over-constrained plans and plans with some parallelism (some sets of tasks can be executed independently). This approach is illustrated with a short example in the next section.

Complete replanning

If a plan repair is not possible, a complete replanning process is mandatory. This problem has not been completely addressed yet, and thus does not appear in the algorithms presented above. The idea is to adapt the approach proposed in [Muscettola *et al.* 1998] ("planning to plan") and consider the planning process as one of the tasks of the plan, in our case a non preemptive task. Thus, the plan on which replanning is started would contain:

- the origin and end horizon timepoints of $ExecutedPlan$,
- the new global state returned by the procedural executive once the execution is completely stopped, associated with the timepoint T set to the reception time,
- a non preemptive task PLAN, with T as start timepoint (and each new timepoint in the plan is constrained to occur after its end timepoint),
- the set of not yet executed goals,
- a new goal requiring the plan to be found.

Open issues remain. One is for instance the detection and abandonment of goals that can not be achieved because of a lack of time.

Example

Let consider a robot with two arms (LH and RH), initially located in $L3$. This robot has to take two objects ($O1$ and $O2$, respectively located in $L1$ and $L2$), and to bring them in $L4$. The robot capabilities are described as a set of four tasks: MOVE from a location to another one, TAKE an object with one of its arms, CARRY the object from a location

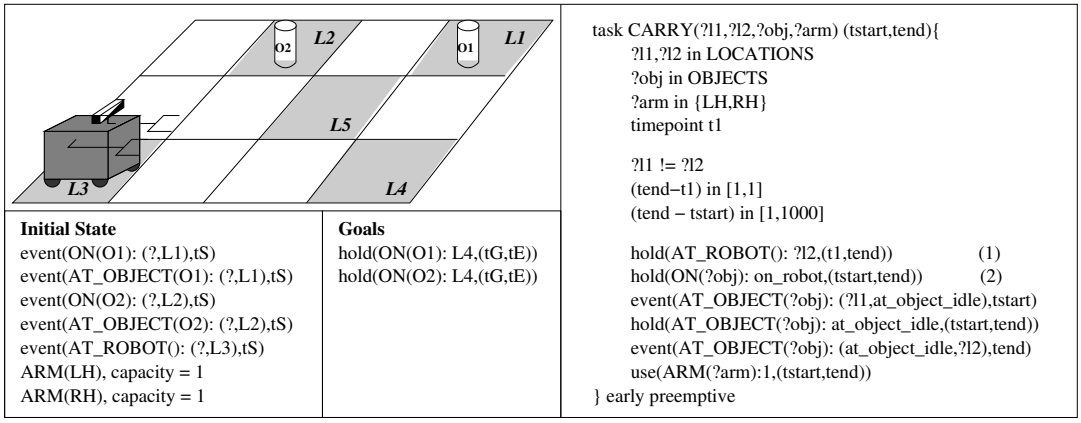

Figure 2: Example of IXTET formalism.

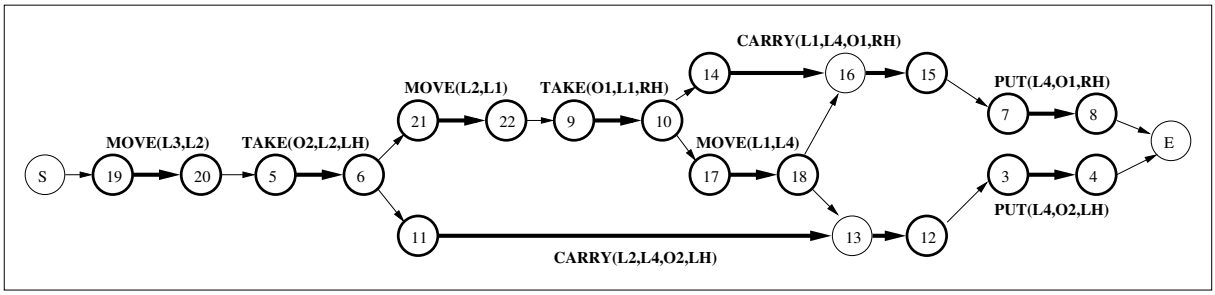

Figure 3: The initial plan.

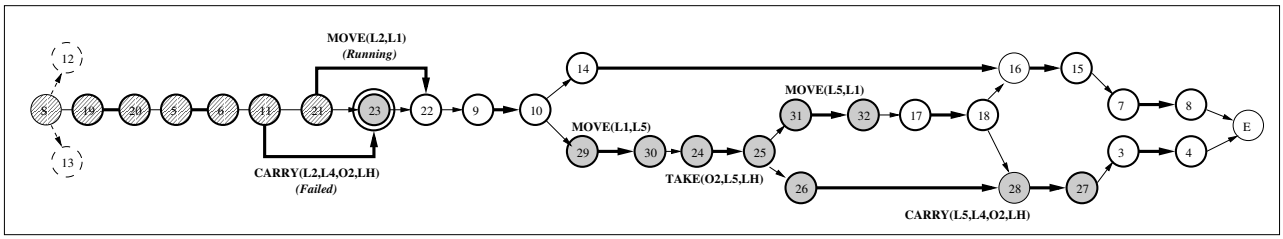

Figure 4: The plan after repair.

to another one and PUT the object. The initial state and goals as well as an example of a task description in the IXTET formalism are illustrated in Figure 2. The CARRY task is early preemptive. It will be terminated as soon as the robot arrives in its final location $?l2$ with the object $?obj$. The proposition (1) asserts that the object is on the robot and the proposition (2) guarantees that the robot is in $?l2$ 1 second before the possible end of the task.

Figure 3 presents the initial plan found by IXTET-EXEC (bold circles represent start and end timepoints of the tasks, arrows represent the precedence constraints between timepoints). The execution starts and a failure occurs: the robot lets $O2$ fall on the floor at the location $L5$ while it is going to $L1$.

Figure 4 presents the plan repaired by IXTET-EXEC. The failure occurred at the beginning of the CARRY task, a failure timepoint (23) has been created and timepoints 12 and 13 relaxed. The part of the plan concerned by the invalidation is related only to the attributes representing the position of $O2$. The task PUT(L4,O2,LH) is no more supported, but the task TAKE(O1,L1,RH) remains valid and can be executed. The shaded timepoints represent the tasks added by the plan repair. Note that this repaired plan is not optimal. Since no initial task has been removed (especially MOVE(L1,L4) is no more useful), the plan contains an extra MOVE from $L5$ to $L1$.

As said before, two parameters are defined by the user: the timestep and how much of it is allocated to the plan repair. Their values mainly depend on the size of the plan. In fact, several factors play a part in the duration of the exe-

cution cycle, among them: the number of timepoints executed during the cycle, the duration of the propagation in the STN (varies with the number n of timepoints in the plan, the complexity is in $(n^2 + n)$), the duration of the plan invalidation in case of failure and the duration of the most expensive planning step. The simple example above has been run on a SunBlade100. Plan invalidation takes 190ms. Plan repair requires 29 steps and 1 backtrack and is distributed on 2 cycles for a 1s timestep (plan repair: 85%), on 4 cycles for a 600ms timestep (plan repair: 80%). During the other cycles, a 5Hz control rate is achieved.

Conclusion and Prospectives

This paper presents some preliminary results on IxTeT-eXeC, an extension to the IxTeT planning system, which is able to interleave more closely planning and temporal execution control. In particular, it regularly updates the plan under execution, it reactively repairs the plan in case of failure, and it incrementally replans upon arrival of new goals.

The process of plan repair in IxTeT-eXeC allows, to some extent, concurrent planning and execution. It is well adapted for domains where subsystems are rather independent and allow some sets of tasks to be executed in parallel. Moreover, this repair technique is "safe" if the domain is such that no failure is fatal, and can always be recovered from. In any case, for critical situations where the system does not have time to repair or replan, one can always consider using predefined emergency plan or procedure, which can be fired by the procedural executive to put the system in a safe state (safe enough to allow a lengthy replanning from scratch).

The work presented here is still ongoing, and we have already identified a number of desirable features, and in some cases, potential methods and solutions to address them:

- We plan to handle uncontrollable timepoints [Morris, Muscettola, & Vidal 2001].
- One of the main advantages of IxTeT is its handling of production, consumption or borrowing of resources. The quantities can be defined as variables ranging over continuous domains. We aim at exploiting this flexibility to update the actual levels of resource during execution, detect future resource contention, repair if possible (add a production task ...) or replan (if a resource is no more available).
- The insertion of new goals is quite similar to the plan repair process. A goal is sent by the procedural executive. It is inserted in the plan as a $hold$ proposition with the adequate temporal constraints. As for plan invalidation, causal links on common attributes are removed to allow the insertion of new tasks and the plan is "repaired" to satisfy the new open condition.

Another important aspect of this work is to embark it and test it on real robotics platforms. Considering that all the other tools and functional modules are currently available on a number of robots at LAAS (Diligent, Dala, etc), and that the development of IxTeT-eXeC is made under Linux (the operating systems used on all these platforms) we do not foresee any particular implementation problem. However, depending on the complexity of the planning task and the dynamic of the environment, we still need to test how well the current implementation will perform on real applications.

References

Dechter, R.; Meiri, I.; and Pearl, J. 1989. Temporal constraint networks. In *KR'89: Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann.

Gaborit, P. 1996. Planification distribuée pour la coopération multi-agents. *Thèse de Doctorat, Université Paul Sabatier, Toulouse*.

Ghallab, M., and Laruelle, H. 1994. Representation and Control in Ixtet, a Temporal Planner. In *Proceedings of the International Conference on AI Planning Systems*, 61–67.

Lemai, S., and Ingrand, F. 2003. Interleaving temporal planning and execution: Ixtet-exec. In *Proceedings of the ICAPS Workshop on Plan Execution*.

Morris, P. H.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. 1998. Remote agent : To boldly go where no ai system has gone before. *Artificial Intelligence* 103.