Local Planning of Ongoing Activities*

Michael Beetz and Drew McDermott
Yale University, Department of Computer Science
P.O. Box 208285, Yale Station
New Haven, CT 06520-8285
Tel.: (203) 432-1229, Fax: (203) 432-0593
beetz@cs.yale.edu, mcdermott@cs.yale.edu

Abstract

An agent that learns about the world while performing its jobs has to plan in a flexible and focused manner: it has to reflect on how to perform its jobs while accomplishing them, focus on critical aspects of important subtasks, and ignore irrelevant aspects of their context. It also has to postpone planning when lacking information, reconsider its course of action when noticing opportunities, risks, or execution failures, and integrate plan revisions smoothly into its ongoing activities.

In this paper, we add constructs to RPL (Reactive Plan Language) that allow for local planning of ongoing activities. The additional constructs constitute an interface between RPL and planning processes that is identical to the interface between RPL and continuous control processes like moving or grasping. The uniformity of the two interfaces and the control structures provided by RPL enable a programmer to concisely specify a wide spectrum of interactions between planning and execution.

Introduction

The problem of how to coordinate planning and execution is central to the effective use of planning to control autonomous robots. It has been studied for a long time, but still remains an open research issue. Typically, autonomous robots that perform their jobs in partly unknown and changing environments acquire important and previously missing information during plan execution. To exploit this information they have to plan in a flexible and focused manner: when they perceive an empty box they should think about whether that box could help them to perform their jobs more efficiently. If the box will be helpful, the robots then need to plan how to use it. Or, for instance, if they see another robot, they should reason about whether the other robot might interfere with some of their plans and revise these plans if necessary. Or, if they are to deliver objects from one location to another, they should postpone planning the errand until they knows how many objects they are to deliver. In all these examples the robot has to respond to risks and opportunities by (re)planning subtasks and postponing planning as long as information is missing. The robot also has to stop planning if the information used for planning is detected to be outdated or install plan revisions without causing problems for the robot's ongoing activities.

We call this kind of planning that flexibly interacts with plan execution *local planning of ongoing activities*. Besides its flexibility, local planning of ongoing activities is also computationally attractive: it allows an embedded robot planning system to plan in realtime by extracting and solving small planning problems on the fly and integrating their solutions smoothly into the robot's overall plan.

We investigate the problem within the XFRM framework, a particular framework for planning and action (McDermott 1992; Beetz & McDermott 1994). The XFRM framework is based on the following principles:

- *Action:* the course of action is specified by *structured reactive plans* that specify how the robot is to respond to sensory input in order to accomplish its jobs. Structured reactive plans are written in RPL (Reactive Plan Language), which provides constructs for sequencing, conditionals, loops, local variables, and subroutines. The language also contains high-level concepts (interrupts, monitors) that can be used to synchronize parallel actions, to make plans reactive, etc.
- *Planning:* XFRM applies a planning technique called *transformational planning of reactive behavior:* the robot controller retrieves routine plans for the robot's jobs from a plan library. Planning processes revise structured reactive plans to avoid flaws in the robot's routine behavior that become predictable as the robot learns new information.

* This work was supported by the Defense Advanced Research Projects Agency, contract number N00014-91-J-1577, administered by the Office of Naval Research.

XFRM and RPL allow for a very flexible and tight integration of planning and execution based on three ideas: (1) Extending RPL with three constructs: one for starting planning processes on subplans, one for installing revisions in a concurrently executed plan, and one for postponing planning of individual subplans; (2) designing the constructs such that they constitute an interface between RPL and planning processes that is identical to the interface between RPL and continuous control processes like moving or grasping; (3) implementing planning processes such that they behave like sensing routines that monitor aspects of the world: whenever something interesting happens during planning, such as a better plan has been found or the search for better plans is completed, the planning process sends a signal. Planning processes provide and continually update their best results in program variables.

Due to the uniformity of the interfaces between RPL/planning processes and RPL/physical control processes and the control structures provided by RPL, a plan writer can concisely specify a wide spectrum of interactions between planning and execution in a behavior-based robot control framework. RPL becomes a single high-level language that can handle both planning and execution actions. Providing the extensions for local planning of ongoing activities as primitives in RPL enables RPL's control structures to control not only the robot's physical actions but also its planning activities. It also allows RPL plans to synchronize threads of plan execution with local planning processes and specify the communication between them.

In the remainder of the paper we describe how local planning capabilities can be integrated into RPL and how important patterns of interaction between planning and execution can be realized in extended RPL.

The Reactive Plan Language (RPL)

We implement plans in RPL (Reactive Plan Language, (McDermott 1991)), a Lisp-like robot control language with conditionals, loops, program variables, processes, and subroutines. RPL provides high-level constructs (interrupts, monitors) to synchronize parallel physical actions and make plans reactive and robust by incorporating sensing and monitoring actions, and reactions triggered by observed events.

RPL also provides concepts that are crucial for the realization of local planning of ongoing activities. The first one is the concept of *concurrency*. Various control structures in RPL create, start, and terminate RPL processes that can run concurrently. RPL also provides *fluents*, program variables that can signal changes of their values. Using fluents a programmer can write pieces of RPL plans that react to asynchronous events. For instance, the statement (WHENEVER *(= f v) r*) is a loop that executes a plan *r* every time the fluent *f* gets the value *v*. The statement (WAIT-FOR *(= f v)*) waits until the fluent *f* gets the value *v*. Fluents can be set by sensing processes, physical control routines, and assignment statements. In the context of runtime planning, RPL control structures give the robot controller the ability to create and control local planning processes and specify the synchronization and communication between planning and execution processes using fluents.

Tasks are another important concept in the implementation of local planning. A *task* is a piece of a plan *pl*, which the system intends to execute. The interpretation of complex tasks creates subtasks. For example, executing the plan $P =$ (N-TIMES 5 (GRASP *ob*)) generates a task *T* with plan (N-TIMES 5 (GRASP *ob*)) and five subtasks *(iter i T)* for $i = 1, \ldots, 5$ with plan (GRASP *ob*). Within the RPL interpreter this notion of task corresponds to a data structure — similar to a stack frame in standard programming languages. This data structure stores the computational state of interpreting *pl*. In RPL plans, tasks are first-class objects and can therefore be passed as arguments of planning and plan swapping processes. The statement (:TAG A (GO 4 5)) statement creates a variable A and sets the value of A to the task "going to location ⟨4,5⟩". Tasks can also be indexed relative to other tasks. The expression (SUBTASK *(iter 3) T*) returns the task for the third iteration of *T* in the example above. Tasks are created on demand, i.e., when they are executed, inspected, referred to, or revised. Some tasks are calls to physical control routines. They start continuous feedback loops that control the robot's effectors and continue immediately with the interpretation of the RPL plan. Control routines use fluents for storing their execution status and the results of their execution and update these fluents automatically. Thus, a piece of RPL plan *p* that calls a control routine *c* and resumes its execution after *c* is done, has to call *c*, wait for the status fluent of *c* to get the value *done*, and then execute the rest of *p*.

The only way RPL plans can affect the execution of control routines is by causing them to *evaporate*. RPL does not provide a command to terminate tasks. Rather the evaporation of tasks is caused by control structures. For instance, the RPL statement (TRY-ALL p_1 ... p_n) creates subtasks for executing $p_1, \ldots, p_n$ and runs them in parallel. The successful execution of any subtask causes the TRY-ALL task to succeed and all other subtasks to evaporate. Evaporation is a dangerous concept because plans and control routines cannot be stopped at any time. A plan for crossing a street shouldn't evaporate the in the midst of its execution.

Using the construct EVAP-PROTECT a programmer can postpone task evaporation until the end of such critical regions. As we will see shortly, we can use the mechanism of task evaporation to swap a new, improved plan for an old one and to terminate local planning processes if further planning is no longer beneficial.

Extending RPL

We have considered two options for the implementation of local planning. First, we investigated extending a planning system to handle multiple planning jobs and enable it to synchronize its planning activities with the rest of the plan. Second, we sought to keep the planning system simple, solve only one planning problem at a time, but provide RPL with means to start and terminate planning processes. The second solution — providing means for creating and controlling simple planning processes — has two advantages. First, existing planning algorithms do not have to be changed a lot to use them for local planning. Second, we can employ RPL to handle the synchronization and communication between planning and execution. RPL is already designed to let the programmer specify how concurrent physical activities of a robot should interact. Thus, it is natural to extend these facilities to mental activities of the robot. We have implemented local planning such that it looks to the RPL interpreter as it were a physical control routine and can be used as a primitive statement in RPL control structures. In this solution we can formulate expressions like the following ones as parts of reactive plans.

WHENEVER Event *ev* happens
replan task *tsk* &
execute revised plan

IF you have a better plan for *tsk*
THEN use the better plan

The RUNTIME-PLAN Statement

The planning algorithms that realize local planning are assumed to have the following functionality: given a task t, the algorithm tries to compute plans that are estimated to be better for performing the task t considering the execution of the plan so far. Each time the algorithm has found a better plan p' it provides p', the best plan it has found so far, as its result. In our experiments we have used planning algorithms that schedule errands (McDermott 1992) and forestall behavior flaws (Beetz & McDermott 1994). Several other planning algorithms have the required functionality (McDermott 1992; Drummond & Bresina 1990; Dean *et al.* 1993; Lyons & Hendriks 1992).

For our implementation of local planning, we have used a simplified version of the XFRM planning algorithm (McDermott 1992; Beetz & McDermott 1994). In XFRM planning is implemented as a search in plan space. A node in the space is a proposed plan; the initial node is a plan that represents the local planning problem. A step in the space requires three phases. First, XFRM projects a plan to generate sample execution scenarios for it. Then, in the *criticism* phase, XFRM examines these execution scenarios to estimate how good the plan is and to identify flaws and summarize them as failure descriptions. XFRM diagnoses the projected execution failures and uses the diagnosis to index into a set of transformation rules that are applied in the third phase, *revision*, to produce new versions of the plan.

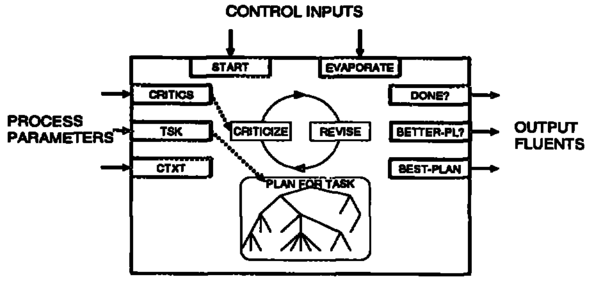

Figure 1: Interface of a planning process.

We extend RPL with the statement (RUNTIME-PLAN *tsk critics best-plan better-plan? done? ctxt*), where *tsk* and *ctxt* are tasks, *critics* is a set of critics, and *best-plan*, *better-plan?*, and *done?* are fluents. The statement starts a planning process that tries to improve the plan for task *tsk* according to the critics *critics* and provides the best plan it has found so far in *best-plan*. *done?* is set to true if the search space has been explored completely. The statement can specify to plan task *tsk* in the context of another task *ctxt*. In order to plan locally, the runtime planner extracts the subplan out of the global plan, revises it using planning, and stores the best plan it has found so far in a fluent. A call to RUNTIME-PLAN creates a new LISP process for locally planning *tsk*. A planning process started by RUNTIME-PLAN is synchronized and communicates with the rest of the plan with fluents described above. Viewed as a black box (see figure 1), the behavior of a planning process is similar to that of a control routine. It can be started and caused to evaporate. It takes the task for which a better plan is to be found and the critics to be used as arguments. The planning process automatically updates three fluents. The fluent *better-plan?* is set true whenever the planner believes that it has found a better plan. The fluent *done?* is false as long as the planner is still looking for better plans. The fluent *best-plan* contains the best plan for the given task that the planner can offer in this moment.

The following example shows a typical use of local planning, how it can be started and caused to evaporate. First, fluents to be updated by the local planning process are created. The body of the LET clause starts two parallel processes: the first one locally plans the task LOCAL-TSK, the root task of the second process. The TRY-ALL statement makes sure that the local planning process evaporates after at most 50 time units or immediately after the task LOCAL-TSK is completed.

```
(LET ((DONE? (CREATE-FLUENT 'DONE? NIL))
      (BETTER-PLAN? (CREATE-FLUENT 'BETTER-PL? NIL))
      (BEST-PLAN (CREATE-FLUENT 'BEST-PLAN NIL)))
  (PAR (TRY-ALL (WAIT-TIME 50)
                (WAIT-FOR (TASK-END LOCAL-TSK))
                (RUNTIME-PLAN LOCAL-TSK CRITICS
                              DONE? BETTER-PL? BEST-PLAN))
       (:TAG LOCAL-TSK (WHAT-EVER))))
```

Ideally, the local planning process should reason about the local plan and the part of the global plan that is relevant for its execution. Thus, the first step in local planning is the construction of a plan p' that resembles the local plan in the context of the global plan closely and is much simpler than the global plan. Of course, it is impossible to construct such a p' for arbitrary reactive plans. To discuss this issue, let us consider the piece of RPL plan in figure 2a. This piece of plan is a LET statement that defines the local variables X and Y and initializes them to 4 and 6. The body of the LET-statement is a statement of the form (WITH-POLICY *pol body*) that specifies that *body* should be executed with *pol* as a constraint. In our example the policy is the control routine AVOID-OBSTACLES and the body contains a subplan with the name A that is supposed to get the robot to location ⟨X,Y⟩. When executing the plan, the robot controller starts to execute the body of the WITH-POLICY statement and whenever the range sensor detects an obstacle close to the robot the policy (AVOID-OBSTACLES) interrupts the execution of the body, navigates the robot away from the obstacle and then resumes the execution of the body.

Now, suppose we want to locally replan the subplan with the tag A. There are several alternatives for determining the local plan and the relevant context of the global plan. The first one is to think only about the local plan, i.e., (ACHIEVE (LOC ROBOT ⟨X,Y⟩)). The advantage of this approach is that it is simple and cheap. The disadvantage is that when projecting the execution of the plan A, the robot might bump into obstacles, an execution failure that cannot occur because the execution of A is constrained by the policy for avoiding obstacles. The other extreme is to always project the whole plan and to ignore plan revisions that are not concerned with the subtask. The advantage of this approach is that the planner does not ignore relevant parts of the plan; the disadvantage is that it is computationally very expensive. A third approach is to consider as relevant the part of the global plan that is executed concurrently. This approach handles nonlocal policies, interrupts, and resources appropriately but cannot handle all global effects of the local plan. For instance, a revision of a local plan to make it faster could block the execution of concurrent subplans and thereby cause the violation of other deadlines.

```
(LET ((X 4) (Y 6))
   (WITH-POLICY                    (LET ((X 4) (Y 6))
       (AVOID-OBSTACLES)              (WITH-POLICY
     ... (PAR ...                            (AVOID-OBSTACLES)
            (:TAG A (GO ⟨X,Y⟩))         (:TAG A (GO ⟨X,Y⟩))))
            ...) ...))
```

Figure 2: Extraction of a local plan. (a) plan. (b) resulting local plan.

Instead of trying to implement algorithms that are able to construct any conceivable planning problem for any given RPL plan we propose that plans should be designed to facilitate the construction of local planning problems. To this end, we have implemented an algorithm that makes the following assumptions about the global plan when constructing a planning problem. First, the relevant information is available at the start of the local planning process. Second, the policies surrounding the local plan and the plan specified in the context argument of RUNTIME-PLAN are the only subplans that strongly interact with the local plan. Making assumptions about plans is far more attractive than making assumptions about the robot and the world because we can engineer the plans such that the assumptions hold. In particular, transforming an RPL plan into another one that satisfies these assumptions is in most cases straightforward.

Under these assumptions a simple algorithm suffices for the construction of local planning problems. The planner constructs a local planning problem by extracting the local plan and embedding it into the policies that constrain its execution in the original plan. If a context for the local plan is specified then the plan of the context task is projected but only failures in the local plan are diagnosed and forestalled. Figure 2b shows the plan constructed for local planning from the original in figure 2a. This approach tends to generate small local planning problems that resemble the original problem well if the assumptions hold.

Plan Swapping

Once an improved plan is found, the robot controller must execute it in place of the original one. We call this process *plan swapping*. The RPL statement (SWAP-PLAN *pl t*) edits the task *t* such that the plan for executing *t* is set to *pl*. To see how plan swapping can be done in RPL consider the following piece of plan that consists of two processes that are executed concurrently. The first process is a sequence of steps with some initial steps α and a final step that calls a RPL procedure FOO. The task for the final step is tagged with the name ACT. The second process consists of the local planning process and a task for swapping plans, and is caused to evaporate by the end of task (FOO). The task for swapping in a new plan wakes up whenever the local planner has found a better plan (the fluent BETTER-PLAN? becomes true) and swaps in the plan stored in the fluent BEST-PLAN as the new plan for ACT.

```
(PAR (SEQ α (:TAG ACT (FOO)))
     (TRY-ALL (:TAG PLAN (RUNTIME-PLAN TSK C DONE?
                                  BETTER-PLAN? BEST-PLAN))
              (WHENEVER BETTER-PLAN?
                  (SWAP-PLAN BEST-PLAN TSK))
              (WAIT-FOR (TASK-END TSK)))
```

The plan swapping mechanism should distinguish three cases: a new plan is found (1) before starting ACT, (2) while executing ACT, or (3) after completing ACT. In the cases (1) and (3) the integration is easy. If the interpreter has not started to execute the subtask the planner simply exchanges the plan text. If the task is already performed the plan does not have to be swapped in at all. The difficult case is when the interpreter has partially executed the plan ACT. Suppose the runtime planner is to swap in a new plan for a partially executed task of the form (N-TIMES 2 *p* UNTIL *c*) after the first iteration. If *p* is picking up an object then redoing the whole statement wouldn't be a problem because the robot would in the worst case try the pickup the object more often than necessary. If however, *p* is "type in the security code of your bankcard" then the robot better keeps track of how often it has already tried to type it in to avoid its bankcard getting withdrawn. In this case, the revised plan has to determine the parts of the original plan which have been executed and skip them.

As for the construction of local planning problems, we assume that plans are designed to facilitate plan swapping. This assumption requires that each task evaporation leaves the robot in a physical state that is safe, and in a computational state that reflects its physical state. The first part of the assumption can be achieved by protecting critical pieces of RPL code against evaporation using the EVAP-PROTECT construct. The second part can be satisfied in different ways. If redoing the steps that have already been executed does not do any harm no precautions have to be taken. If there are points in plan execution at which plan swapping is safe we can synchronize execution and plan swapping such that plans are swapped only at these points. For instance, in most cases it is safe to swap the body of a loop between two iterations. In the remaining cases the execution of the revised plan should ideally pick up where the original subplan stopped, which can be assured in two ways. First, the state of plan execution can be determined by sensing. In the other case the original plan has to store the state of execution in global variables and update these variables as execution proceeds. The revised plan has to decide which of its subplans to skip based on the values of these variables. We call a plan *p* ***restartable*** if for any task *t* that has *p* as its plan, starting the execution of *p*, causing its evaporation, and executing *p* completely afterwards causes a behavior that satisfies the task *t*.

Thus, we can require of RPL plans that each task *t* for which a step (SWAP-PLAN *p′ t*) can be executed concurrently has a plan *p* that is restartable. Under the assumption that all plans satisfy this property, we can implement the plan swapping algorithm in a simple way: delete all the current subtasks of *t*, and create a new subtask to execute *p′*. If *t* is active, we cause all of *t*'s earlier subtasks to evaporate and start *t* anew.

Making Planning Assumptions

The planning system makes planning assumptions to simplify planning problems. We have added the statement (WHEN-PROJECTING *asmp body*) that tells the plan projector to generate projections for *body* that satisfy the assumption *asmp*. An assumption can be of the form *(SUCCEEDS tsk)*, which asserts that task *tsk* is assumed to succeed. If the projector projects *tsk* it uses a causal model that specifies the effects of a successful execution of *tsk*. If *tsk* has the form (ACHIEVE *(HAS ROBOT OB)*), the effects are not only that the robot has the object *OB* in its hand, but also that the robot is at the same location as *OB*, knows that *OB* is in its hand, that the robot's hand force sensor signals pressure in its hand, and so on.

We use planning assumptions for three purposes. First, we use them to speed up local planning processes by ignoring unimportant aspects of the context of tasks. Second, we reduce the uncertainty that has to be considered by the planning system; instead of probabilistically choosing the value of a random variable the planner can explore hypothetical situations and answer questions like "how would the delivery plan work if sensing is perfect?" Finally, to avoid projecting local planning processes we specify the expected effects of local planning processes using planning assumptions.

We do not want the plan projector to project RUNTIME-PLAN statements for two reasons. First, projecting a RUNTIME-PLAN-statement is computationally very expensive, about as expensive as executing it. Secondly, the purpose of local planning is often to deal with unexpected information, and therefore, it is unlikely that we can generate this information probabilistically. Thus, the projector ignores RUNTIME-PLAN statements and the programmer asserts the expected effects of local planning using planning assumptions.

Suppose the agent has a plan (ACHIEVE *g*) that needs to be replanned in the case of rain. Replanning can be done by wrapping a policy around the achievement step. As soon as the robot notices rain, the body of the WITH-POLICY statement is interrupted and a local planning process is started. This process runs until it finds a better plan, which is swapped in before the execution of the body is resumed. Since a global planning process is supposed to assume that the local planning process takes care of the contingency that it is raining, we have to assert that, in the case of rain, the goal (ACHIEVE *g*) will succeed.

Implementing Deliberative Controllers

We will now apply the constructs we have introduced in the previous section to implement several important patterns of interactions between planning and execution. The examples include a self-adapting iterative plan, a plan that monitors assumptions and forestalls execution failures if the assumptions are violated, and a plan that recovers from detected execution failures by replanning.

Execution monitoring and replanning to avoid execution failures. Because of imperfect world and causal models used for planning, the execution of a plan might not proceed as predicted by the planner. Usually the planner predicts for each stage of execution a set of future states and makes planning decisions for the later course of action that depend on some of these states. In this case, the success of future steps depends on the truth of states at earlier stages of execution. Thus, the plan can avoid future execution failures by actively sensing the truth of these states and reconsidering its course of action if they are false. Checking these states is part of a process called *execution monitoring*.

Processes which monitor vital states and signal their violation through fluents can be used for this kind of execution monitoring. The rest of the plan contains subplans that react to unwanted changes in the state, for instance, by replanning or reachieving the state. We can synchronize the monitoring step with the rest of the plan by specifying the relative importance (priority) of monitoring and execution, that monitoring is only done if the camera is unused, or by restricting the temporal scope of the monitoring task, etc.

The following piece of RPL code shows a delivery plan that assumes the absence of other robots interfering with the delivery. A global policy is wrapped around the plan that watches out for other robots whenever the camera is not used by other tasks. If the policy detects another robot it sets the fluent OTHER-ROBOT-AROUND*. The policy around the delivery plan replans the delivery under consideration of the possibly interfering robot whenever the fluent is pulsed.

```
(WITH-POLICY
   (LET ((PERCEIVED-ROBOTS '()))
      (LOOP (WAIT-FOR (NOT (IN-USE CAMERA)))
             (!= PERCEIVED-ROBOTS (LOOK-FOR ((CAT ROBOT))))
             (IF (NOT (NULL PERCEIVED-ROBOTS))
                (PULSE OTHER-ROBOT-AROUND*))))
   ...
   (WITH-POLICY (WHENEVER OTHER-ROBOT-AROUND*
                     (RUNTIME-PLAN DELIVERY ...))
      (:TAG DELIVERY ...)))
```

Improving iterative plans by local planning. The following piece of RPL code is a loop that can adapt its behavior to changing situations by locally planning the next iteration of the loop. The loop body consists of two subplans that are performed concurrently. The first subplan controls the behavior of the robot in the current iteration. The second subplan tries to improve the plan for the next iteration. The local planning process starts with the plan for the current iteration. The planning process is caused to evaporate at the end of each iteration. If a better plan is found, the better plan is swapped in for the next iteration.

```
(:TAG L
   (LOOP
      (!= I (+ I 1))
      (PAR (:TAG AN-ITER (DEFAULT-ITER))
           (LET ((NEXT-ITER
                   (PATH-SUB
                     ((TAGGED AN-ITER) (ITER (+ I 1))) L)))
             (SEQ (TRY-ALL (SEQ (SWAP-PLAN (TASK-PLAN AN-ITER)
                                            NEXT-ITER)
                                 (RUNTIME-PLAN
                                    NEXT-ITER ... BETTER-PLAN
                                    ... BEST-PLAN)
                                 (WAIT-FOR BETTER-PLAN?)
                           (WAIT-FOR (TASK-END AN-ITER)))
                  (IF BETTER-PLAN?
                     (SWAP-PLAN BEST-PLAN NEXT-ITER)))))))
```

Variants of this kind of planned iterative behavior have been implemented for the kitting robot (Lyons & Hendriks 1992) and for controlling the walking behavior of Ambler (Simmons 1990).

Recovering from execution failures. Sometimes execution failures cannot be forestalled and therefore local planning processes have to plan to recover from execution failures after they have occurred. Plans can often recognize execution failures, such as a missed grasping action, by checking whether the pressure measured by the hand force sensor after grasping is still zero. This execution failure together with a failure description can be generated by appending (IF (= HAND-FORCE-SENSOR* 0) (FAIL :CLASS NOTHING-GRASPED)) to the grasping routine. Failures and their descriptions are passed to supertasks and cause supertasks to fail unless they have a failure handling capability. For instance, the macro (WITH-FAILURE-HANDLER *handler body*) can recover from a failure in *body* by executing *handler* with the failure description as its argument. Suppose a delivery task signals a *perceptual confusion error* because more than one object has satisfied a given object description. In this case, the handler starts a local plan revision process for the delivery plan. The revised plan is then executed to recover from the failure. If the failure recovery is not successful an irrecoverable failure is signalled to the super tasks.

```
(WITH-FAILURE-HANDLER
     (λ (FAILURE)
        (IF (IS perceptual-confusion FAILURE)
            (TRY-IN-ORDER
               (SEQ (RUNTIME-PLAN ERR-REC
                        :CRITICS '(PERCEPTUAL-CONFUSION)
                        :TIME-RESOURCE 20)
                    (:TAG ERR-REC
                        (ACHIEVE '(LOC ,DES (0,8)))))
               (FAIL :CLASS IRRECOVERABLE-FAILURE FAILURE)))
            (FAIL :CLASS IRRECOVERABLE-FAILURE FAILURE)))
  (:TAG BODY (ACHIEVE '(LOC ,DES (0,8)))))
```

Some robot control architectures. Beside common patterns of interaction between planning and execution we can also implement agent architectures in extended RPL. The implementation of the Sense/Plan/Act architecture (Nilsson 1984) is straightforward. It consists of a loop with three steps: The first step of an iteration comprises the sense and model steps, the second step is a call to runtime plan, the third step is the execution that is started after planning is completed, i.e., the *done?* fluent is set.

A more interesting example is a simplified version of XFRM's agent architecture. In XFRM, planning and execution are performed in parallel. Whenever the planner has found a better plan, it sets the fluent NEW-PLAN* and thereby causes the current plan to evaporate. The first step of the loop body terminates if a better plan is found or WHOLE-PLAN completely executed. If now the subplan tagged WHOLE-PLAN is done then the loop terminates; otherwise an improved plan is swapped in and the next iteration started.

```
(:TAG XFRM
    (PAR (:TAG PLANNER
             (LOOP (RUNTIME-PLAN WHOLE-PLAN ...)))
         (:TAG CONTROLLER
             (LOOP (TRY-ALL
                      (SEQ (WAIT-FOR NEW-PLAN*)
                           (!= NEW-PLAN* NIL))
                      (:TAG WHOLE-PLAN ...))
               UNTIL (DONE? WHOLE-PLAN)
                  (SWAP-PLAN BEST-PLAN WHOLE-PLAN)
```

Conclusion

Locally planning ongoing activities will be an important capability of future service robots that perform their jobs in changing and partly unknown environments. The research reported in this paper makes important contributions to the realization of such capabilities. The main contribution is a single high-level language that coordinates planning and execution actions. To implement the language we have started with a robot control/plan language that offers powerful control abstractions for concurrent plan execution, handling asynchronous events, and causing control processes to evaporate. In addition, the language had to provide stack frames created during plan interpretation as computational objects that can be passed as arguments to local planning and plan swapping processes. We have added local planning capabilities as primitive statements that start planning processes and swap plans. The protocol for interacting with local planning processes has been designed is the same as the one for physical control routines, which enables control programs to control physical and planning actions in exactly the same way. The primitives for starting planning processes and plan swapping have built-in mechanisms for constructing local planning problems and revising partially executed plans.

Besides the language extensions we contribute solutions for two computational problems that are essential for the coordination of planning and execution: the revision of partially executed plans and the construction of local planning problems. We have shown that plans can be designed so that simple algorithms can solve these two problems robustly and efficiently.

The ideas underlying local planning of ongoing activities and the success of our realization are difficult to evaluate. Our realization of local planning required minimal extensions of an existing language for behavior-based robot control: only three statements were added. The extended control language enables plan writers to specify a wide range of very flexible interactions between planning and execution. We have seen in the previous section that, using the language, existing robot control architectures can be implemented cleanly and concisely. The examples we gave also showed that planning and physical actions have to synchronized in the same way as physical actions. Thus, the control abstractions for synchronizing physical actions are equally useful for specifying the interaction between planning and execution. In our experience, programming the interaction between planning and execution using the control structures provided by RPL made the implementation of these control architectures particularly easy.

We have successfully implemented variants of the code pieces discussed in the previous section and used them to control a simulated robot in a simulated world. In particular, we have embedded a robot controller in a simulated robot performing a varying set of complex tasks in a changing and partly unknown environment. The controller locally plans the ongoing activities of the robot. Thus whenever the robot detects a contingency, a planning process is started that projects the effects of the contingency on the robot's plan and — if necessary — revises the plan to make it more robust. Using this controller, the robot performs its jobs almost as efficiently as it would using efficient default plans, but much more reliably (see (Beetz 1996)).

Related Work. Any robot that executes its plans has to coordinate its planning and execution actions. Most systems implement a particular type of coordination between planning and execution. Others leave the coordination flexible. Our approach proposes to specify the interaction between planning and execution as part of the robot controllers and provides the necessary control statements.

PLANEX (Fikes, Hart, & Nilsson 1972) and SIPE (Wilkins 1988) reason to recover from execution failures. NASL (McDermott 1978) and IPEM (Ambros-Ingerson & Steel 1988) interleave planning and execution. NASL uses planning to execute complex actions and IPEM integrates partial-order planning and plan execution. $\mathcal{RS}$ (Lyons & Hendriks 1992) integrates planning and execution for highly repetitive tasks. The interactions between planning and action in these systems can be expressed concisely in extended RPL.

Hybrid-layered control systems (for example, (Gat 1992; Bonasso, Antonisse, & Slack 1992)) plan and execute in parallel by decoupling planning and execution with a sequencing layer. The most common approach for the implementation of the sequencing layer is the RAP system (Firby 1989) that can create, manipulate, and manage networks of sketchy plan steps with different priorities. In RPL, a successor of RAP, we can accomplish the same kinds of behavior by implementing vital behavior as global policies that are always active. The functionality of the sequencing layer is provided by the operating system of RPL that manages and updates the task queue according to the semantics of RPL control structures. PRS (Georgeff & Ingrand 1989) and blackboard architectures (Hayes-Roth 1989) are data-driven control systems that use "meta-level" reasoning to choose between different applicable courses of action. The RPL extensions for local planning of ongoing behavior are tools that facilitate the implementation of such "meta-level" reasoning modules and their synchronization with reactive execution.

Our work is closely related to Simmons' TCA architecture (Simmons 1990), which provides control concepts to synchronize different threads of control, interleave planning and execution, recover from execution failures, and monitor the environment continuously. RPL provides control structures for the same purposes at a higher level of abstraction. We are currently trying to compile RPL programs into programs for TCA.

Acknowledgments. We would like to thank the reviewers for their insightful comments.

References

Ambros-Ingerson, J., and Steel, S. 1988. Integrating planning, execution, and monitoring. In *Proc. of AAAI-88*, 735–740.

Beetz, M., and McDermott, D. 1994. Improving robot plans during their execution. In Hammond, K., ed., *Proc. of AIPS-94*, 7–12.

Beetz, M. 1996. *Anticipating and Forestalling Execution Failures in Structured Reactive Plans*. Technical report, Yale University. forthcoming.

Bonasso, P.; Antonisse, H.; and Slack, M. 1992. A reactive robot system for find and fetch tasks in an outdoor environment. In *Proc. of AAAI-92*.

Dean, T.; Kaelbling, L.; Kirman, J.; and Nicholson, A. 1993. Planning with deadlines in stochastic domains. In *Proc. of AAAI-93*, 574–579.

Drummond, M., and Bresina, J. 1990. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proc. of AAAI-90*, 138–144.

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4):189–208.

Firby, J. 1989. *Adaptive Execution in Complex Dynamic Worlds*. Technical report 672, Yale University, Department of Computer Science.

Gat, E. 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proc. of AAAI-92*.

Georgeff, M., and Ingrand, F. 1989. Decision making in an embedded reasoning system. In *Proc. of the 11 th IJCAI*, 972–978.

Hayes-Roth, B. 1989. Intelligent monitoring and control. In *Proc. of the 11 th IJCAI*, 243–249.

Lyons, D., and Hendriks, A. 1992. A practical approach to integrating reaction and deliberation. In *Proc. of AIPS-92*, 153–162.

McDermott, D. 1978. Planning and acting. *Cognitive Science* 2(2):71–109.

McDermott, D. 1991. A reactive plan language. Research Report YALEU/DCS/RR-864, Yale University.

McDermott, D. 1992. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University.

Nilsson, N. 1984. Shakey the robot. Technical Note 323, SRI AI Center.

Simmons, R. 1990. An architecture for coordinating planning, sensing, and action. In Sycara, K., ed., *Innovative Approaches to Planning, Scheduling and Control*, 292–297. San Mateo, CA: Morgan Kaufmann.

Wilkins, D. 1988. *Practical Planning: Extending the AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann.