Planning for Reactive Robot Behavior

D. M. Lyons and A. J. Hendriks

Philips Laboratories
North American Philips Corporation
Briarcliff Manor NY 10510
dml@philabs.philips.com

Abstract

A robot system operating in an environment in which there is uncertainty and change needs to combine the ability to react with the ability to plan ahead. In a previous paper we proposed a solution to the problems of integrating planning and reaction: cast planning as *adaptation* of a reactive system. In this paper, we extend our theoretical treatment of the planner, including the effect of its iterated adaptation on the reactor, and also describe our current implementation environment and a working example.

1 Introduction

We have previously introduced a practical problem, the kitting robot problem [?], that necessitates integrating *reactive, real-time* responses with the ability to *plan ahead 'on the fly'*. It has proven difficult to integrate these capabilities: Classical AI planning [?] is not sufficiently robust in uncertain and dynamic environments[?]. Purely reactive approaches [?] operate very robustly in some uncertain and dynamic environments — namely those for which the reactions have been manually pre-programmed – but lack any ability to plan ahead. In [?] we proposed a solution to this integration: cast planning as *adaptation* of a reactive system.

In this paper we take this work further. We extend our theoretical treatment of the planner-reactor system to include the effects of iterated adaptations by the planner on the reactor. This forces us to introduce some novel ideas such as planner *expectation,* explicit planner perception requests, and a modular description of a problem space, *situations.* To balance this theoretical treatment, we describe our implementation and work through a planner-reactor example we have currently running.

We begin with an overview of the relevant literature in Section 2. Section 3 reviews the planner-reactor architecture. Section 4 presents our new results, showing how the concepts of planner expectation and situations are necessary to define the convergence of the reactor. We have had the planner-reactor system in operation on a Puma-560 for over a year to date, and in Section 5 we describe this implementation and work through an example kitting task we have implemented.

2 Literature Review

There have been a number of approaches to integrating planning and reaction (for a summary, see [?]). The most common approach has been to 'piggy-back' a planner on top of a reactive scheduler/executer, e.g., SROMA [?] for robotic assembly, and e.g., [?] for robot motion planning. The concept there is that an abstract 'plan' is generated *a-priori*, and then fed to a reactive executer which is allowed to modify action ordering or introduce other deviations. We argue the resultant system behavior is only partially reactive because the generation and loading of entire plans is simply not feasible for most dynamic and uncertain environments, and such a reactive scheduler/executer can only 'react' once it has been issued with a plan.

Gatt [?] has discussed connecting a reactive system to a high-level planner. In his approach, the planner sits hierarchically on top of the reactive system, and can inhibit or allow certain behaviors. This is, however, insufficient, since in general the planner needs to be able to also introduce new/unique behaviors, and there needs to be support for characterizing repeated planner interaction with the reactive system.

In [?], we introduced an approach which cast planning as the on-going *adaptation* of a reactive system to suit changes in goals or the environment. Rather than generating and then executing plans, our approach is to have the reactor be a real-time system that continually interacts with the environment, and the planner be a separate and concurrent system that 'tunes' the behavior of the reactor to ensure that goals are achieved. This eliminates the problems of reactive-scheduling, since the reactor is at all times simply a reactive system. It also eliminates the problems of the hierarchical planner-reactive system approach: it allows the introduction of novel behaviors and facilitates describing repeated planner interactions.

3 Planner-Reactor Overview

This section very briefly summarizes the planner-reactor approach from [?]. The reactor is a system containing a network of sensory-motor 'linkages' which we will call reactions. It has two key properties: Unlike a plan executor, it can produce action at *any* time, and it is a *real-time* system. The reactor is initialized with a number of generally useful behaviors already in place.

The planner is completely separate from, and concur-

Figure 1: Planner-Reactor-World System

rent with, the reactor (Figure 1). It continually cycles, determining if the reactor's behavior in the current environment will indeed conform to the planners objectives. If it determines that the reactor will not achieve these objectives, it makes an incremental change to the reactor configuration to bring its behavior back into line.

As indicated in Figure 1, there are two connections between planner and reactor:

1. *Perceptions:* Some of the sensory data gathered by the reactor is fed into the planner (the planner controls which data).
2. *Adaptations:* The planner alters the structure of the reactor by specifying adaptations.

3.1 The $\mathcal{RS}$ Model

The $\mathcal{RS}$ model is a formal, process-based model of computation developed for robot programming. This model allows us to formalize the structure of the reactor and to capture the mechanism of reactor adaptation. A subset of the model has also been implemented as a robot programming language. The following is a brief overview of $\mathcal{RS}$, for a detailed motivation and description see [?].

An $\mathcal{RS}$ description of a process, or network of processes, is called a *schema*. For example, $\mathtt{Joint}_{j=v}$ denotes a process that is an instance of the schema `Joint` with parameter j set to value v (when unambiguous, we write $\mathtt{Joint}_v$). Processes can be composed into larger networks using several kinds of *process composition operators*. At the lowest level, all networks have to be compositions of a set of atomic, pre-defined, processes. The composition operators are: *sequential* `A;B`, *concurrent* `(A,B)`, *conditional* `A:B` and *disabling* `A#B`. There are two non-atomic, recurrent composition operators: *synchronous recurrent* $\mathtt{A} :; \mathtt{B} = \mathtt{A} : (\mathtt{B}; \mathtt{A} :; \mathtt{B})$, and *asynchronous recurrent* $\mathtt{A} :; \mathtt{B} = \mathtt{A} : (\mathtt{B}, \mathtt{A} :; \mathtt{B})$.

It is important to represent how process networks evolve over time, as processes dynamically terminate or are created. This is captured by the evolves operator: process `A` evolves into process `B` under condition Ω if `A` becomes equal to `B` when condition Ω occurs, written as $\mathtt{A} \xrightarrow{\Omega} \mathtt{B}$. Evolves is axiomitized from the definitions of the composition operators, e.g., $\mathtt{A};\mathtt{B} \xrightarrow{\Omega\mathtt{A}} \mathtt{B}$.

3.2 Reactor

The reactor `R` consists of a set of sensory-action pairs, reactions, and a well-defined interface through which the planner can modify its structure.

$$\mathtt{R} = (\ \mathtt{E}^1\#\mathtt{P}^1, \ldots, \mathtt{E}^n\#\mathtt{P}^n,\ \mathtt{ADDPLAN}\langle k\rangle :: (\mathtt{E}^k\#\mathtt{P}^k)\ ) \qquad (1)$$

where

1. $\mathtt{P}^i$ is a sensory-action pair or a reactor segment.
2. $\mathtt{E}^i$ is a guard process through which $\mathtt{P}^i$ can be removed by the planner (by the #-composition).
3. `ADDPLAN` is the interface through which the planner can add new guarded reactions $\mathtt{E}^m\#\mathtt{P}^m$ (by the ::-composition).

Reactions, hard-wired responses to sensory conditions, are specified in $\mathcal{RS}$ as compositions of sensory and motor processes. A reactor does not contain/execute plans in the same sense that a classical plan executer/scheduler does, i.e., as a data structure that is manipulated or executed. Nonetheless, certain parts of the reactor structure can be identified as corresponding to the instructions necessary for a specific task or subtask. We refer to these as *reactor segments.*

3.3 Adaptation

We can write down how the reactor is affected by the changes described in items 2 and 3 above

$$\begin{array}{ccc} (\ \mathtt{R}\ ,\mathtt{E}^k\#\mathtt{P}^k\ ) & \xrightarrow{Dk} & \mathtt{R} \\ \mathtt{R} & \xrightarrow{Ak} & (\ \mathtt{R}\ ,\mathtt{E}^m\#\mathtt{P}^m\ ) \end{array} \qquad (2)$$

$$\begin{array}{l} where\ Dk\ is\ \mathtt{E}^k \longrightarrow \mathtt{STOP}, and \\ where\ Ak\ is\ \mathtt{ADDPLAN}\langle k\rangle \longrightarrow \mathtt{STOP}\&k = m \end{array}$$

That is Dk can be used to *delete* reactions and Ak can be used to *add* reactions, and the process of reactor adaptation can be expressed as:

$$\mathtt{R} \xrightarrow{\alpha} \mathtt{R}' \qquad (3)$$

where $\mathtt{R}'$ is the adapted reactor and α is some combination of Ak and Dk conditions. This adaptation mechanism describes the way in which the planner can modify the reactor to improve its behavior.

4 The Planner

Planning is a continual process of accepting goals from a user and perceptions from the reactor, and producing adaptations to the reactor.

We assume the planner has the following kinds of knowledge in its Environment Model, EM: A domain theory, DT, that contains knowledge about causality and likelihood in the problem domain[1]; An action repertoire, a description of the actions available to the reactor; A world model, a collection of currently known facts about the world; and, the reactor model, the description of the reactor currently in operation.

The Ideal Reactor. The planner must have as its objective the construction of a reactor that achieves the goals G not only in the current environment, but because change is possible, in all environments that might occur, as indicated by the environment model. A reactor that fulfills this criterion is called an *ideal reactor,* and is similar in behavior to Schoppers universal plans [?]. For a given goal G and environment model EM , we write the the ideal reactor as the $\mathcal{RS}$ process $\mathtt{R}^*_{EM,G}$. The behavior of the ideal reactor can be captured with the evolves operator. For the ideal reactor, only evolutions of the following form are possible:

$$(\ \mathtt{R}^*\ ,\ \mathtt{EM}\ ) \longrightarrow (\ \mathtt{R}^*\ ,\ \mathtt{GEM}\ ) \tag{4}$$

That is, when the ideal reactor is run concurrently with a process implementing the environment model the only possible evolutions are those leading to an environment in which the goal is satisfied GEM .

The ideal reactor is the most robust reactor possible, however, it is likely to be a very large and complex machine. It is unrealistic to presuppose that the planner will always be able to generate the ideal reactor in one step. Most likely it won't be able to do so for the following reasons:

1. It may not have sufficient time because of the time constraints on actions in the environment.
2. There may be too much uncertainty in the world model for the reactor to be constructed without doing some more sensing.
3. It may not be possible to built the ideal reactor because of resource and action constraints.

Thus, the ideal reactor is an eventual target for the planner, but since it cannot be constructed all at once, an equally important issue is finding a method that constructs the ideal reactor in 'useful' increments.

Incremental construction of the ideal reactor. Incremental changes to the reactor should result in another working reactor which is capable of behavior at least as goalworthy as the the original reactor. The ideal reactor definition above (4) can be restricted with conditions:

$$(\ \mathtt{R},\ \mathtt{EM}\ ) \xrightarrow{\omega} (\ \mathtt{R},\ \mathtt{GEM}\ ) \tag{5}$$

This says that as long as assumptions ω hold, then R behaves like the ideal reactor. We call this the ω-ideal reactor $\mathtt{R}^{\omega}_{EM,G}$ for environment EM and goals G.

As it stands, this claims nothing about the behavior of the reactor if the assumptions don't hold. However, in the case of failed assumptions, it is necessary that the reactor do nothing that would prohibit a future improved version of the reactor from operating successfully. Therefore our reactor segment formulation [?] insists that all reactions are guarded by assumption monitors. Thus when an assumption fails, all the reactions that use this assumption are prevented from interacting (inappropriately) with the environment.

Using the likelihood of ω holding in the environment as the ordering criterion on ω enables the planner to build reactor adaptations to deal with relaxation of less likely assumptions first. The evolution of the reactor under adaptation (3) can therefore be rewritten as:

$$\begin{array}{c}\mathtt{R}^{\omega 1} \xrightarrow{\alpha 1} \mathtt{R}^{\omega 2} \xrightarrow{\alpha 2} \mathtt{R}^{\omega 3} \xrightarrow{\alpha 3} \ldots \xrightarrow{\alpha n} \mathtt{R}^* \\ for\ \ \omega 1 > \omega 2 > \omega 3 > \ldots > \emptyset\end{array} \tag{6}$$

where αi is some combination of Ak and Dk conditions. Thus, the planner always has as its starting point a particular reactor R and a set of assumptions ω, and its objective is to produce the ω-ideal reactor. It is unrealistic to believe this can be done in one increment, for the same reasons that the ideal reactor can't be produced in one increment. We need to investigate how R is adapted to become $\mathtt{R}^{\omega}$.

Any reactor R can be seen as a part of the ideal reactor, since we can always choose a set of reactions that inhibit reactions in R not in $\mathtt{R}^{\omega}$ and add in reactions in $\mathtt{R}^{\omega}$ missing from R. That is, for any $\mathtt{R}^{\omega}$ we can say:

$$\mathtt{R}^{\omega} = (\ \mathtt{R}^{\omega 1}\ ,\ \bar{\mathtt{R}}\ )^c \tag{7}$$

Where $\bar{\mathtt{R}}$ contains the extra information described. In [?], we theorized that $\bar{\mathtt{R}}$ could be used as a measure of how close R was to $\mathtt{R}^*$, and we suggested an ordering for $\bar{\mathtt{R}}$. The process of incremental construction now boils down to updating R over time so that $\bar{\mathtt{R}}$ is reduced to null using a combination of Ak and Dk conditions. We now address how to come up with this reduction.

Adaptation Increments. At every planner iteration t, based on the environment model EM_t and current goals G_t, the planner generates what we call an *expectation*, E_t; an abstract description of the changes it expects to make to the reactor to achieve G_t. At every iteration, the combination of the reactor model R_t and the expectation E_t is exactly the ω-ideal reactor (where ω is the current set of assumptions the planner is working with):

$$R_t + E^{\omega}_t = R^{\omega}_t \tag{8}$$

This is the planner's version of (7) above, where now $\bar{\mathtt{R}}$ can be given meaning as the implementation of the planners current expectation E_t for the reactor. The process of incremental improvement of the reactor is the incremental reduction of the planner's expectation by some ΔE_t and corresponding increment of the reactor ΔR_t, until when E_t is null, the current reactor is exactly the ideal reactor.

Situations. In [?] we introduced a concept called *situations.* We claim it is a good modular unit to handle adaptation increments. Formally, situations are equivalence relations on the state-space for a specific task in a specific environment. Intuitively, a situation being active expresses the appropriateness for the reactor to undertake the set of reactions associated with that situation. Many situations may be active at one time.

[1] The DT may contain uncertain information, and information about dynamic events. However, the DT is assumed to be *correct* and *invariant* over time. A changing/incorrect DT would necessitate a learning component, and we are as yet only interested in only the planning problem.

Figure 2: Planner-Reactor Interactions.

Figure 3: The Planner Architecture.

Situations have the following useful characteristics. They lend themselves to the use of reactions: Within a situation, the planner doesn't really care what sequence the reactor takes to achieve the goal. Thus, the actions in a situation can be handled by a set of reactions keyed to fire in that situation. Situations facilitate incremental construction: The situation defines a unit of useful activity, so the planner can reduce its expectation in units of situations. Situations facilitate relaxation of assumptions: The relaxation of an assumption will always result in either addition of one or more new situations, (reflecting the extra work needed in the absence of the assumption), or the addition of additional reflexes into a situation (if removal of the assumption can be covered by a generalized behavior within the same situation), or both.

The Full System. The input/output interactions between planner and reactor are as shown in figure 2. The planner incrementally produces changes in the reactor structure ΔR_t plus the associated assumption monitors ΔI_t, and additional perception processes ΔP_t. It receives back information from existing perception process VP_t, and any signals from assumption monitors that have failed FI_t. The planner architecture is described in figure 3. Based on the environment model EM and the goals G_t, the planner accumulates a current problem-solving context. Within that PSC the planner generates an abstract plan, the expectation E_t. The PSC also provides the current set of assumptions $Assm_t$. This set is used to filter the environment model to generate a simpler model. The current expectation E_t is analysed to determine how much of it can be reduced ΔE_t. This incremental expectation is decomposed (in the planning sense) in the context of this filtered world, to generate an adaptation of the reactor ΔR_t, and the set of assumption monitoring processes to go along with it ΔI_t. If further sensing is necessary to decrease E_t then a set of perception requests ΔP_t is also generated.

Convergence to the ideal reactor. Roughly, the convergence problem for a planner-reactor is defined as follows: Given a specific goal and initial reactor show that eventually the reactor will be improved to the ideal reactor. in the world. This question concerns the evolves chain in formula (6). For the moment we do not attempt to give a formal answer to the issue of convergence, but rather we use the above evolves chain to discuss the constraints on convergence. Convergence takes place in two loops: in the internal loop an expectation E_t based on a specific set of assumptions $Assm_t$ is gradually reduced to zero and the reactor increased to contain the plan for that expectation. $Assm_t$ is then reduced by the least likely assumption(s), and the inside loop again activated.

Consider the outer loop first. In this loop, if an invariant failure occurs, then the set of assumptions will be prematurely relaxed. For convergence to occur in a worst-case world, it would have to occur despite repeated assumption monitor failure. Let us assume a finite set of assumptions. In that, this convergence constraints just says that if `R` is to converge in a worst case world, then it must be possible to relax the assumptions in any order and still arrive at the ideal reactor.

In the internal loop, in order to reduce E_t to zero, the following must hold: Firstly, it must be possible to reduce the expectation by finite increments (otherwise it never gets smaller). Secondly, if there is uncertainty in EM_t critical to reducing E_t, it must either be possible to produce perception processes that eventually clear up the uncertainty, or else have reactions that operate despite the uncertainty.

5 Implementation Example

5.1 Environment

We have constructed a robot workcell to explore the planner-reactor approach in the kitting domain. Figure 4 shows the basic layout of our implementation, described briefly below. For more detail on implementation see [?] and also our tape in the video proceedings. A Puma-560 robot equipped with a four-fingered force-sensing gripper is the basis of the kitting workcell. The workcell has two cameras. The VAL controller is connected by a serial line to one of a set of processors on a common VMEBUS. This proces-

Figure 4: The Kitting Robot Implementation.

Figure 5: Kitting Example Situations.

sor can issue commands to the robot in two modes: a coarse position mode (VAL `MOVE`) and a fine position/velocity mode (VAL `ALTER`). The two cameras are connected to a Philips PAPS industrial vision system. This system is controlled by another of the processors on the common VME-BUS. A `recognize` command causes this server to scan the image from one of the cameras and determine if there are any instances of a given set of 2D object models in the image.

$\mathcal{RS}$ System. A subset of the $\mathcal{RS}$ model has been implemented as a robot programming language, `RS-L3`. It uses threaded-execution to implement concurrency A set of basic sensory and motor schemas has been built to interface to the robot and vision servers. For example, $\mathtt{Locate}_m$ is a basic sensory process that continually queries the vision server with `recognize(m)` commands and only terminates when such a model instance is seen. The reactor is a set of `RS-L3` process definition equations, as described in eqn. (1). The `E`, `I` and `ADDPLAN` processes are all basic processes, and all communicate with a remote planner using UNIX socket links.

Planner. The planner uses Pelavin's[?] Interval Temporal Logic (ITL) formalism to represent world models. and rules. in a network and new relations are enforced by constraint propagation [?]. Formulas in ITL are first order logic predicates augmented with a modal operator @, e.g., $IN(A, D)@i1$ specifies that the predicate $IN(A, D)$ should be evaluated over interval $i1$. In addition the special predicate $ITL(interval1, itl - relation, interval2)$ is used to specify constraints between intervals, e.g., $ITL(i1, equal, i2)$ specifies that the interval $i1$ should denote the same interval of time as $i2$. Other relations we use here include $finishes$ (both finish at the same point) and $encompasses$ (the second is a subinterval of the first).

5.2 Kitting Example

Consider the following kitting robot example: The robot looks for a kit tray t, fills in parts $p1, p2$ and $p3$ (which arrive via a conveyer belt) on that tray, and then sends the tray out of the cell.

New Goals. The goal in this example is

$$\begin{array}{l} \forall t, i1 : Instance(tray, t)@i1 \Rightarrow \\ \quad \exists i2 : (Filled(t) \wedge At(t, buffer))@i2 \wedge \\ \quad\quad ITL(i2, finishes, time) \end{array} \tag{9}$$

This conditional goal specifies that whenever there is an instance of a tray, then that that tray should be filled with parts and placed in the output buffer. The ITL predicate in the consequent assures that, once assembled, the kit stays together by requiring the goal interval $i2$ to finish the special interval $time$, denoting the entire time line. The goal $Filled$ can be written out as:

$$\begin{array}{c} Filled(t)@i1 \iff \\ [\ \exists x, y, z Instance(p1, x) \wedge Instance(p2, y) \\ \wedge Instance(p3, z) \wedge Contains(t, x, y, z)\ ]@i1 \end{array} \tag{10}$$

Problem Solving Context. The planner constructs a PSC, its'frame of mind', in which it is going plan for this goal. In our domain theory, goals are associated with relevant actions and domain rules. The planner accumulates this information in the PSC. For example, $Instance$ brings in potentially useful actions such as `Locate` and `Acquire` ; $Contains$ brings in a number of subgoals, such as $In(t, x)$. In addition, the domain rules may specify default 'normalness' statements about the world, i.e., predicates that are potentially useful assumptions. In this example, the only assumption brought in is caused by `Acquire`. A precondition for $\mathtt{Acquire}_x$ is that we have $static(x)$, i.e., that x not be a moving object. Thus, although the planner does not know $static(x)$, it asserts it in the current PSC, and includes it in the set of assumptions.

Regressing on the conditional goal in this PSC gives us the first abstract description of the reactor segment, E_t (see fig 5(a)): first locate any tray, then fill that tray, and finally remove the tray to the output buffer.

$$\begin{array}{l} \mathtt{P1} = \mathtt{NoTray} :; [\ \mathtt{Locate}_{tray}\langle t\rangle : (\mathtt{Acquire}_t ; \mathtt{Put}_{t,ws})\ ]\#\mathtt{I}_{static} \\ \mathtt{P2} = \mathtt{Tray}\langle t\rangle :; \mathtt{STUB}_{t,stub1} \\ \mathtt{P3} = \mathtt{Filled}\langle t\rangle :; \mathtt{STUB}_{t,stub2} \end{array}$$

The situation monitors, `NoTray`, `Tray` and `Filled`, use sensory data to determine which situation holds (per tray).

The first situation has an accompanying reaction – to watch out for any tray visually (`Locate`) and if and when one is found to bring it into the workspace ws. The reaction is protected by an assumption monitor $\mathtt{I}_{static}$ to warn the planner if the $static(x)$ assumption has been violated. The other situations are known, so the planner can be warned if they are fired (`STUB`), but no actions are provided yet. This segment is added in as follows: The definitions `P1-P3` are loaded into the $\mathcal{RS}$ system. The planner then instructs the `ADDPLAN` process of the current reactor to terminate with $k = 1, 2$ and then 3, adding in the new segments.

Refinement. The expectation can now be decreased by the situation $NoTray$. On the next refinement the $tray$ situation is decomposed. Given that the parts can be placed in any order, the state space of the problem is shown in figure 5(b). Applying our definition of situations simplifies this to the situation space shown in figure 5(c), with the relation between states and situations as indicated. This results in the following reactor segment:

$$\begin{aligned}
&\mathtt{P4} = \mathtt{LacksP1}\langle t\rangle ::;\\
&\quad [\, \mathtt{Locate}_{p1}\langle p\rangle : (\mathtt{Acquire}_p ; \mathtt{Insert}_{p,t}) \,]\#\mathtt{I}_{static}\\
&\mathtt{P5} = \mathtt{LacksP2}\langle t\rangle ::;\\
&\quad [\, \mathtt{Locate}_{p2}\langle p\rangle : (\mathtt{Acquire}_p ; \mathtt{Insert}_{p,t}) \,]\#\mathtt{I}_{static}\\
&\mathtt{P6} = \mathtt{LacksP3}\langle t\rangle ::;\\
&\quad [\, \mathtt{Locate}_{p3}\langle p\rangle : (\mathtt{Acquire}_p ; \mathtt{Insert}_{p,t}) \,]\#\mathtt{I}_{static}
\end{aligned}$$

Where the three situations are `LacksP1`, `LacksP2` and `LacksP3`, indicating, in this case, that the tray t is *still missing* the corresponding part. Each reaction is guarded by an assumption monitor. This segment is added into the existing reaction as follows: the definitions are loaded into the $\mathcal{RS}$ system, the $\mathtt{E}_2$ process is instructed to terminate, removing the $tray$ situation, and `ADDPLAN` is instructed to terminate with $k = 4, 5$ and then 6.

The addition of the final situation segment $Filled$ proceeds in the same fashion.

Forced Refinement. Consider now the case that one of the assumption monitors fails, indicating that the robot has interacted with a moving object. Say this occurred on part $P2$. That reaction is immediately disabled (by #-composition) by the failure of its $\mathtt{I}_{static}$ monitor and the planner notified. The planner now removes $static(x)$ from its assumption set, and sets it's value to false in its current PSC (since it now knows its value). Since the set of assumptions has been changed, a new expectation is generated.

First the planner considers the segment that was directly affected (and in immediate need of attention therefore). Thus, the expectation now again contains the situation $LacksP2$. In our example, there is another action `Track` which allows the robot to acquire a moving object. Decomposing the expectation in this updated PSC produces the new reflex:

$$\mathtt{P7} = \mathtt{LacksP1}\langle t\rangle ::; \mathtt{Locate}_{p1}\langle p\rangle : (\mathtt{Track}_p ; \mathtt{Insert}_{p,t})$$

The reactor is updated by commanding $\mathtt{E}_5$ to terminate, and commanding `ADDPLAN` to terminate with $k = 7$. Subsequently, the planner will work to remove the $static(x)$ assumption from all the situations.

6 Discussion

This paper builds upon our reactor description in [?] to describe the objectives and internal structure of the planner component. In particular we take a closer look at the iterative effects of planner adaptation actions. In basic design, the planner-reactor falls into the class of what in adaptive control are called *self-tuning regulators*. $\mathcal{RS}$ provides us with a powerful discrete-event control representation: more concise for this problem than automaton or petri-net models, and yet with some mathematical tools (such as the evolves operator) available. We believe therefore that it is possible to construct a stronger theoretical treatment of the iterative effects of planner adaptation. A stronger theory is also necessary to address issues of the observability of assumptions and of what constitutes a good initial reactor.

Planning for Reactive Robot Behavior

D. M. Lyons and A. J. Hendriks

Philips Laboratories
North American Philips Corporation
Briarcliff Manor NY 10510
dml@philabs.philips.com

Abstract

A robot system operating in an environment in which there is uncertainty and change needs to combine the ability to react with the ability to plan ahead. In a previous paper we proposed a solution to the problems of integrating planning and reaction: cast planning as *adaptation* of a reactive system. In this paper, we extend our theoretical treatment of the planner, including the effect of its iterated adaptation on the reactor, and also describe our current implementation environment and a working example.

1 Introduction

We have previously introduced a practical problem, the kitting robot problem [9], that necessitates integrating *reactive, real-time* responses with the ability to *plan ahead 'on the fly'*. It has proven difficult to integrate these capabilities: Classical AI planning [3] is not sufficiently robust in uncertain and dynamic environments[10]. Purely reactive approaches [2] operate very robustly in some uncertain and dynamic environments — namely those for which the reactions have been manually pre-programmed – but lack any ability to plan ahead. In [9] we proposed a solution to this integration: cast planning as *adaptation* of a reactive system.

In this paper we take this work further. We extend our theoretical treatment of the planner-reactor system to include the effects of iterated adaptations by the planner on the reactor. This forces us to introduce some novel ideas such as planner *expectation,* explicit planner perception requests, and a modular description of a problem space, *situations.* To balance this theoretical treatment, we describe our implementation and work through a planner-reactor example we have currently running.

We begin with an overview of the relevant literature in Section 2. Section 3 reviews the planner-reactor architecture. Section 4 presents our new results, showing how the concepts of planner expectation and situations are necessary to define the convergence of the reactor. We have had the planner-reactor system in operation on a Puma-560 for over a year to date, and in Section 5 we describe this implementation and work through an example kitting task we have implemented.

2 Literature Review

There have been a number of approaches to integrating planning and reaction (for a summary, see [8]). The most common approach has been to 'piggy-back' a planner on top of a reactive scheduler/executer, e.g., SROMA [12] for robotic assembly, and e.g., [4] for robot motion planning. The concept there is that an abstract 'plan' is generated *a-priori*, and then fed to a reactive executer which is allowed to modify action ordering or introduce other deviations. We argue the resultant system behavior is only partially reactive because the generation and loading of entire plans is simply not feasible for most dynamic and uncertain environments, and such a reactive scheduler/executer can only 'react' once it has been issued with a plan.

Gatt [5] has discussed connecting a reactive system to a high-level planner. In his approach, the planner sits hierarchically on top of the reactive system, and can inhibit or allow certain behaviors. This is, however, insufficient, since in general the planner needs to be able to also introduce new/unique behaviors, and there needs to be support for characterizing repeated planner interaction with the reactive system.

In [9], we introduced an approach which cast planning as the on-going *adaptation* of a reactive system to suit changes in goals or the environment. Rather than generating and then executing plans, our approach is to have the reactor be a real-time system that continually interacts with the environment, and the planner be a separate and concurrent system that 'tunes' the behavior of the reactor to ensure that goals are achieved. This eliminates the problems of reactive-scheduling, since the reactor is at all times simply a reactive system. It also eliminates the problems of the hierarchical planner-reactive system approach: it allows the introduction of novel behaviors and facilitates describing repeated planner interactions.

3 Planner-Reactor Overview

This section very briefly summarizes the planner-reactor approach from [9]. The reactor is a system containing a network of sensory-motor 'linkages' which we will call reactions. It has two key properties: Unlike a plan executor, it can produce action at *any* time, and it is a *real-time* system. The reactor is initialized with a number of generally useful behaviors already in place.

The planner is completely separate from, and concur-

Figure 1: Planner-Reactor-World System

rent with, the reactor (Figure 1). It continually cycles, determining if the reactor's behavior in the current environment will indeed conform to the planners objectives. If it determines that the reactor will not achieve these objectives, it makes an incremental change to the reactor configuration to bring its behavior back into line.

As indicated in Figure 1, there are two connections between planner and reactor:

1. *Perceptions:* Some of the sensory data gathered by the reactor is fed into the planner (the planner controls which data).
2. *Adaptations:* The planner alters the structure of the reactor by specifying adaptations.

3.1 The $\mathcal{RS}$ Model

The $\mathcal{RS}$ model is a formal, process-based model of computation developed for robot programming. This model allows us to formalize the structure of the reactor and to capture the mechanism of reactor adaptation. A subset of the model has also been implemented as a robot programming language. The following is a brief overview of $\mathcal{RS}$, for a detailed motivation and description see [6].

An $\mathcal{RS}$ description of a process, or network of processes, is called a *schema*. For example, $\texttt{Joint}_{j=v}$ denotes a process that is an instance of the schema `Joint` with parameter j set to value v (when unambiguous, we write $\texttt{Joint}_v$). Processes can be composed into larger networks using several kinds of *process composition operators*. At the lowest level, all networks have to be compositions of a set of atomic, pre-defined, processes. The composition operators are: *sequential* `A;B`, *concurrent* `(A,B)`, *conditional* `A:B` and *disabling* `A#B`. There are two non-atomic, recurrent composition operators: *synchronous recurrent* $\texttt{A} :; \texttt{B} = \texttt{A} : (\texttt{B}; \texttt{A} :; \texttt{B})$, and *asynchronous recurrent* $\texttt{A} :; \texttt{B} = \texttt{A} : (\texttt{B}, \texttt{A} :; \texttt{B})$.

It is important to represent how process networks evolve over time, as processes dynamically terminate or are created. This is captured by the evolves operator: process `A` evolves into process `B` under condition Ω if `A` becomes equal to `B` when condition Ω occurs, written as $\texttt{A} \xrightarrow{\Omega} \texttt{B}$. Evolves is axiomitized from the definitions of the composition operators, e.g., $\texttt{A}; \texttt{B} \xrightarrow{\Omega\texttt{A}} \texttt{B}$.

3.2 Reactor

The reactor `R` consists of a set of sensory-action pairs, reactions, and a well-defined interface through which the planner can modify its structure.

$$\texttt{R} = (\ \texttt{E}^1\#\texttt{P}^1, \ldots, \texttt{E}^n\#\texttt{P}^n,\ \texttt{ADDPLAN}\langle k\rangle :: (\texttt{E}^k\#\texttt{P}^k)\ ) \qquad (1)$$

where

1. $\texttt{P}^i$ is a sensory-action pair or a reactor segment.
2. $\texttt{E}^i$ is a guard process through which $\texttt{P}^i$ can be removed by the planner (by the #-composition).
3. `ADDPLAN` is the interface through which the planner can add new guarded reactions $\texttt{E}^m\#\texttt{P}^m$ (by the ::-composition).

Reactions, hard-wired responses to sensory conditions, are specified in $\mathcal{RS}$ as compositions of sensory and motor processes. A reactor does not contain/execute plans in the same sense that a classical plan executer/scheduler does, i.e., as a data structure that is manipulated or executed. Nonetheless, certain parts of the reactor structure can be identified as corresponding to the instructions necessary for a specific task or subtask. We refer to these as *reactor segments.*

3.3 Adaptation

We can write down how the reactor is affected by the changes described in items 2 and 3 above

$$\begin{array}{ccc} (\ \texttt{R}\ , \texttt{E}^k\#\texttt{P}^k\ ) & \xrightarrow{Dk} & \texttt{R} \\ \texttt{R} & \xrightarrow{Ak} & (\ \texttt{R}\ , \texttt{E}^m\#\texttt{P}^m\ ) \end{array} \qquad (2)$$

$$\begin{array}{l} \textit{where } Dk \textit{ is } \texttt{E}^k \longrightarrow \texttt{STOP}, \textit{and} \\ \textit{where } Ak \textit{ is } \texttt{ADDPLAN}\langle k\rangle \longrightarrow \texttt{STOP}\&k = m \end{array}$$

That is Dk can be used to *delete* reactions and Ak can be used to *add* reactions, and the process of reactor adaptation can be expressed as:

$$\texttt{R} \xrightarrow{\alpha} \texttt{R}' \qquad (3)$$

where $\texttt{R}'$ is the adapted reactor and α is some combination of Ak and Dk conditions. This adaptation mechanism describes the way in which the planner can modify the reactor to improve its behavior.

4 The Planner

Planning is a continual process of accepting goals from a user and perceptions from the reactor, and producing adaptations to the reactor.

We assume the planner has the following kinds of knowledge in its Environment Model, EM: A domain theory, DT, that contains knowledge about causality and likelihood in the problem domain[1]; An action repertoire, a description of the actions available to the reactor; A world model, a collection of currently known facts about the world; and, the reactor model, the description of the reactor currently in operation.

The Ideal Reactor. The planner must have as its objective the construction of a reactor that achieves the goals G not only in the current environment, but because change is possible, in all environments that might occur, as indicated by the environment model. A reactor that fulfills this criterion is called an *ideal reactor,* and is similar in behavior to Schoppers universal plans [11]. For a given goal G and environment model $\mathtt{EM}$, we write the the ideal reactor as the $\mathcal{RS}$ process $\mathtt{R}^*_{EM,G}$. The behavior of the ideal reactor can be captured with the evolves operator. For the ideal reactor, only evolutions of the following form are possible:

$$(\ \mathtt{R}^*\ ,\ \mathtt{EM}\ ) \longrightarrow (\ \mathtt{R}^*\ ,\ \mathtt{GEM}\ ) \tag{4}$$

That is, when the ideal reactor is run concurrently with a process implementing the environment model the only possible evolutions are those leading to an environment in which the goal is satisfied $\mathtt{GEM}$.

The ideal reactor is the most robust reactor possible, however, it is likely to be a very large and complex machine. It is unrealistic to presuppose that the planner will always be able to generate the ideal reactor in one step. Most likely it won't be able to do so for the following reasons:

1. It may not have sufficient time because of the time constraints on actions in the environment.
2. There may be too much uncertainty in the world model for the reactor to be constructed without doing some more sensing.
3. It may not be possible to built the ideal reactor because of resource and action constraints.

Thus, the ideal reactor is an eventual target for the planner, but since it cannot be constructed all at once, an equally important issue is finding a method that constructs the ideal reactor in 'useful' increments.

Incremental construction of the ideal reactor. Incremental changes to the reactor should result in another working reactor which is capable of behavior at least as goalworthy as the the original reactor. The ideal reactor definition above (4) can be restricted with conditions:

$$(\ \mathtt{R},\ \mathtt{EM}\ ) \stackrel{\omega}{\longrightarrow} (\ \mathtt{R},\ \mathtt{GEM}\ ) \tag{5}$$

This says that as long as assumptions ω hold, then $\mathtt{R}$ behaves like the ideal reactor. We call this the ω-ideal reactor $\mathtt{R}^{\omega}_{EM,G}$ for environment $\mathtt{EM}$ and goals G.

As it stands, this claims nothing about the behavior of the reactor if the assumptions don't hold. However, in the case of failed assumptions, it is necessary that the reactor do nothing that would prohibit a future improved version of the reactor from operating successfully. Therefore our reactor segment formulation [9] insists that all reactions are guarded by assumption monitors. Thus when an assumption fails, all the reactions that use this assumption are prevented from interacting (inappropriately) with the environment.

Using the likelihood of ω holding in the environment as the ordering criterion on ω enables the planner to build reactor adaptations to deal with relaxation of less likely assumptions first. The evolution of the reactor under adaptation (3) can therefore be rewritten as:

$$\begin{gathered}\mathtt{R}^{\omega 1} \xrightarrow{\alpha 1} \mathtt{R}^{\omega 2} \xrightarrow{\alpha 2} \mathtt{R}^{\omega 3} \xrightarrow{\alpha 3} \ldots \xrightarrow{\alpha n} \mathtt{R}^* \\ for\ \ \omega 1 > \omega 2 > \omega 3 > \ldots > \emptyset\end{gathered} \tag{6}$$

where αi is some combination of Ak and Dk conditions. Thus, the planner always has as its starting point a particular reactor $\mathtt{R}$ and a set of assumptions ω, and its objective is to produce the ω-ideal reactor. It is unrealistic to believe this can be done in one increment, for the same reasons that the ideal reactor can't be produced in one increment. We need to investigate how $\mathtt{R}$ is adapted to become $\mathtt{R}^{\omega}$.

Any reactor $\mathtt{R}$ can be seen as a part of the ideal reactor, since we can always choose a set of reactions that inhibit reactions in $\mathtt{R}$ not in $\mathtt{R}^{\omega}$ and add in reactions in $\mathtt{R}^{\omega}$ missing from $\mathtt{R}$. That is, for any $\mathtt{R}^{\omega}$ we can say:

$$\mathtt{R}^{\omega} = (\ \mathtt{R}^{\omega 1}\ ,\ \bar{\mathtt{R}}\ )^c \tag{7}$$

Where $\bar{\mathtt{R}}$ contains the extra information described. In [9], we theorized that $\bar{\mathtt{R}}$ could be used as a measure of how close $\mathtt{R}$ was to $\mathtt{R}^*$, and we suggested an ordering for $\bar{\mathtt{R}}$. The process of incremental construction now boils down to updating $\mathtt{R}$ over time so that $\bar{\mathtt{R}}$ is reduced to null using a combination of Ak and Dk conditions. We now address how to come up with this reduction.

Adaptation Increments. At every planner iteration t, based on the environment model EM_t and current goals G_t, the planner generates what we call an *expectation*, E_t; an abstract description of the changes it expects to make to the reactor to achieve G_t. At every iteration, the combination of the reactor model R_t and the expectation E_t is exactly the ω-ideal reactor (where ω is the current set of assumptions the planner is working with):

$$R_t + E^{\omega}_t = R^{\omega}_t \tag{8}$$

This is the planner's version of (7) above, where now $\bar{\mathtt{R}}$ can be given meaning as the implementation of the planners current expectation E_t for the reactor. The process of incremental improvement of the reactor is the incremental reduction of the planner's expectation by some ΔE_t and corresponding increment of the reactor ΔR_t, until when E_t is null, the current reactor is exactly the ideal reactor.

Situations. In [9] we introduced a concept called *situations.* We claim it is a good modular unit to handle adaptation increments. Formally, situations are equivalence relations on the state-space for a specific task in a specific environment. Intuitively, a situation being active expresses the appropriateness for the reactor to undertake the set of reactions associated with that situation. Many situations may be active at one time.

[1] The DT may contain uncertain information, and information about dynamic events. However, the DT is assumed to be *correct* and *invariant* over time. A changing/incorrect DT would necessitate a learning component, and we are as yet only interested in only the planning problem.

Figure 2: Planner-Reactor Interactions.

Figure 3: The Planner Architecture.

Situations have the following useful characteristics. They lend themselves to the use of reactions: Within a situation, the planner doesn't really care what sequence the reactor takes to achieve the goal. Thus, the actions in a situation can be handled by a set of reactions keyed to fire in that situation. Situations facilitate incremental construction: The situation defines a unit of useful activity, so the planner can reduce its expectation in units of situations. Situations facilitate relaxation of assumptions: The relaxation of an assumption will always result in either addition of one or more new situations, (reflecting the extra work needed in the absence of the assumption), or the addition of additional reflexes into a situation (if removal of the assumption can be covered by a generalized behavior within the same situation), or both.

The Full System. The input/output interactions between planner and reactor are as shown in figure 2. The planner incrementally produces changes in the reactor structure ΔR_t plus the associated assumption monitors ΔI_t, and additional perception processes ΔP_t. It receives back information from existing perception process VP_t, and any signals from assumption monitors that have failed FI_t. The planner architecture is described in figure 3. Based on the environment model EM and the goals G_t, the planner accumulates a current problem-solving context. Within that PSC the planner generates an abstract plan, the expectation E_t. The PSC also provides the current set of assumptions $Assm_t$. This set is used to filter the environment model to generate a simpler model. The current expectation E_t is analysed to determine how much of it can be reduced ΔE_t. This incremental expectation is decomposed (in the planning sense) in the context of this filtered world, to generate an adaptation of the reactor ΔR_t, and the set of assumption monitoring processes to go along with it ΔI_t. If further sensing is necessary to decrease E_t then a set of perception requests ΔP_t is also generated.

Convergence to the ideal reactor. Roughly, the convergence problem for a planner-reactor is defined as follows: Given a specific goal and initial reactor show that eventually the reactor will be improved to the ideal reactor. in the world. This question concerns the evolves chain in formula (6). For the moment we do not attempt to give a formal answer to the issue of convergence, but rather we use the above evolves chain to discuss the constraints on convergence. Convergence takes place in two loops: in the internal loop an expectation E_t based on a specific set of assumptions $Assm_t$ is gradually reduced to zero and the reactor increased to contain the plan for that expectation. $Assm_t$ is then reduced by the least likely assumption(s), and the inside loop again activated.

Consider the outer loop first. In this loop, if an invariant failure occurs, then the set of assumptions will be prematurely relaxed. For convergence to occur in a worst-case world, it would have to occur despite repeated assumption monitor failure. Let us assume a finite set of assumptions. In that, this convergence constraints just says that if `R` is to converge in a worst case world, then it must be possible to relax the assumptions in any order and still arrive at the ideal reactor.

In the internal loop, in order to reduce E_t to zero, the following must hold: Firstly, it must be possible to reduce the expectation by finite increments (otherwise it never gets smaller). Secondly, if there is uncertainty in EM_t critical to reducing E_t, it must either be possible to produce perception processes that eventually clear up the uncertainty, or else have reactions that operate despite the uncertainty.

5 Implementation Example

5.1 Environment

We have constructed a robot workcell to explore the planner-reactor approach in the kitting domain. Figure 4 shows the basic layout of our implementation, described briefly below. For more detail on implementation see [7] and also our tape in the video proceedings. A Puma-560 robot equipped with a four-fingered force-sensing gripper is the basis of the kitting workcell. The workcell has two cameras. The VAL controller is connected by a serial line to one of a set of processors on a common VMEBUS. This proces-

Figure 4: The Kitting Robot Implementation.

Figure 5: Kitting Example Situations.

sor can issue commands to the robot in two modes: a coarse position mode (VAL `MOVE`) and a fine position/velocity mode (VAL `ALTER`). The two cameras are connected to a Philips PAPS industrial vision system. This system is controlled by another of the processors on the common VMEBUS. A `recognize` command causes this server to scan the image from one of the cameras and determine if there are any instances of a given set of 2D object models in the image.

$\mathcal{RS}$ System. A subset of the $\mathcal{RS}$ model has been implemented as a robot programming language, `RS-L3`. It uses threaded-execution to implement concurrency A set of basic sensory and motor schemas has been built to interface to the robot and vision servers. For example, $\texttt{Locate}_m$ is a basic sensory process that continually queries the vision server with `recognize(m)` commands and only terminates when such a model instance is seen. The reactor is a set of `RS-L3` process definition equations, as described in eqn. (1). The `E, I` and `ADDPLAN` processes are all basic processes, and all communicate with a remote planner using UNIX socket links.

Planner. The planner uses Pelavin's[10] Interval Temporal Logic (ITL) formalism to represent world models. and rules. in a network and new relations are enforced by constraint propagation [1]. Formulas in ITL are first order logic predicates augmented with a modal operator @, e.g., $IN(A,D)@i1$ specifies that the predicate $IN(A,D)$ should be evaluated over interval $i1$. In addition the special predicate $ITL(interval1, itl - relation, interval2)$ is used to specify constraints between intervals, e.g., $ITL(i1, equal, i2)$ specifies that the interval $i1$ should denote the same interval of time as $i2$. Other relations we use here include *finishes* (both finish at the same point) and *encompasses* (the second is a subinterval of the first).

5.2 Kitting Example

Consider the following kitting robot example: The robot looks for a kit tray t, fills in parts $p1, p2$ and $p3$ (which arrive via a conveyer belt) on that tray, and then sends the tray out of the cell.

New Goals. The goal in this example is

$$\begin{array}{l} \forall t, i1 : Instance(tray, t)@i1 \Rightarrow \\ \quad \exists i2 : (Filled(t) \wedge At(t, buffer))@i2 \wedge \\ \qquad ITL(i2, finishes, time) \end{array} \tag{9}$$

This conditional goal specifies that whenever there is an instance of a tray, then that that tray should be filled with parts and placed in the output buffer. The ITL predicate in the consequent assures that, once assembled, the kit stays together by requiring the goal interval $i2$ to finish the special interval *time*, denoting the entire time line. The goal *Filled* can be written out as:

$$\begin{array}{c} Filled(t)@i1 \iff \\ [\ \exists x, y, z Instance(p1, x) \wedge Instance(p2, y) \\ \wedge Instance(p3, z) \wedge Contains(t, x, y, z)\ ]@i1 \end{array} \tag{10}$$

Problem Solving Context. The planner constructs a PSC, its'frame of mind', in which it is going plan for this goal. In our domain theory, goals are associated with relevant actions and domain rules. The planner accumulates this information in the PSC. For example, *Instance* brings in potentially useful actions such as `Locate` and `Acquire` ; *Contains* brings in a number of subgoals, such as $In(t,x)$. In addition, the domain rules may specify default 'normalness' statements about the world, i.e., predicates that are potentially useful assumptions. In this example, the only assumption brought in is caused by `Acquire.` A precondition for $\texttt{Acquire}_x$ is that we have $static(x)$, i.e., that x not be a moving object. Thus, although the planner does not know $static(x)$, it asserts it in the current PSC, and includes it in the set of assumptions.

Regressing on the conditional goal in this PSC gives us the first abstract description of the reactor segment, E_t (see fig 5(a)): first locate any tray, then fill that tray, and finally remove the tray to the output buffer.

$$\begin{array}{l} \texttt{P1} = \texttt{NoTray} :; [\ \texttt{Locate}_{tray}\langle t\rangle : (\texttt{Acquire}_t ; \texttt{Put}_{t,ws})\ ] \# \texttt{I}_{static} \\ \texttt{P2} = \texttt{Tray}\langle t\rangle :; \texttt{STUB}_{t,stub1} \\ \texttt{P3} = \texttt{Filled}\langle t\rangle :; \texttt{STUB}_{t,stub2} \end{array}$$

The situation monitors, `NoTray, Tray` and `Filled,` use sensory data to determine which situation holds (per tray).

The first situation has an accompanying reaction – to watch out for any tray visually (`Locate`) and if and when one is found to bring it into the workspace ws. The reaction is protected by an assumption monitor $\mathtt{I}_{static}$ to warn the planner if the $static(x)$ assumption has been violated. The other situations are known, so the planner can be warned if they are fired (`STUB`), but no actions are provided yet. This segment is added in as follows: The definitions `P1-P3` are loaded into the $\mathcal{RS}$ system. The planner then instructs the `ADDPLAN` process of the current reactor to terminate with $k = 1, 2$ and then 3, adding in the new segments.

Refinement. The expectation can now be decreased by the situation $NoTray$. On the next refinement the $tray$ situation is decomposed. Given that the parts can be placed in any order, the state space of the problem is shown in figure 5(b). Applying our definition of situations simplifies this to the situation space shown in figure 5(c), with the relation between states and situations as indicated. This results in the following reactor segment:

$$\begin{aligned}&\mathtt{P4} = \mathtt{LacksP1}\langle t\rangle \text{ :;}\\&\quad [\,\mathtt{Locate}_{p1}\langle p\rangle : (\mathtt{Acquire}_p; \mathtt{Insert}_{p,t})\ ]\#\mathtt{I}_{static}\\&\mathtt{P5} = \mathtt{LacksP2}\langle t\rangle \text{ :;}\\&\quad [\,\mathtt{Locate}_{p2}\langle p\rangle : (\mathtt{Acquire}_p; \mathtt{Insert}_{p,t})\ ]\#\mathtt{I}_{static}\\&\mathtt{P6} = \mathtt{LacksP3}\langle t\rangle \text{ :;}\\&\quad [\,\mathtt{Locate}_{p3}\langle p\rangle : (\mathtt{Acquire}_p; \mathtt{Insert}_{p,t})\ ]\#\mathtt{I}_{static}\end{aligned}$$

Where the three situations are `LacksP1, LacksP2` and `LacksP3,` indicating, in this case, that the tray t is *still missing* the corresponding part. Each reaction is guarded by an assumption monitor. This segment is added into the existing reaction as follows: the definitions are loaded into the $\mathcal{RS}$ system, the $\mathtt{E}_2$ process is instructed to terminate, removing the $tray$ situation, and `ADDPLAN` is instructed to terminate with $k = 4, 5$ and then 6.

The addition of the final situation segment $Filled$ proceeds in the same fashion.

Forced Refinement. Consider now the case that one of the assumption monitors fails, indicating that the robot has interacted with a moving object. Say this occurred on part $P2$. That reaction is immediately disabled (by #-composition) by the failure of its $\mathtt{I}_{static}$ monitor and the planner notified. The planner now removes $static(x)$ from its assumption set, and sets it's value to false in its current PSC (since it now knows its value). Since the set of assumptions has been changed, a new expectation is generated.

First the planner considers the segment that was directly affected (and in immediate need of attention therefore). Thus, the expectation now again contains the situation $LacksP2$. In our example, there is another action `Track` which allows the robot to acquire a moving object. Decomposing the expectation in this updated PSC produces the new reflex:

$$\mathtt{P7} = \mathtt{LacksP1}\langle t\rangle \text{ :;} \mathtt{Locate}_{p1}\langle p\rangle : (\mathtt{Track}_p; \mathtt{Insert}_{p,t})$$

The reactor is updated by commanding $\mathtt{E}_5$ to terminate, and commanding `ADDPLAN` to terminate with $k = 7$. Subsequently, the planner will work to remove the $static(x)$ assumption from all the situations.

6 Discussion

This paper builds upon our reactor description in [9] to describe the objectives and internal structure of the planner component. In particular we take a closer look at the iterative effects of planner adaptation actions. In basic design, the planner-reactor falls into the class of what in adaptive control are called *self-tuning regulators*. $\mathcal{RS}$ provides us with a powerful discrete-event control representation: more concise for this problem than automaton or petri-net models, and yet with some mathematical tools (such as the evolves operator) available. We believe therefore that it is possible to construct a stronger theoretical treatment of the iterative effects of planner adaptation. A stronger theory is also necessary to address issues of the observability of assumptions and of what constitutes a good initial reactor.

References

[1] J.F. Allen. Towards a general theory of action and time. *Artificial Intell.*, 23(2):123–154, 1984.

[2] R. Brooks. A robust layered control system for a mobile robot. *IEEE J. Rob. & Aut.*, RA-2(1):14–22, Mar. 1986.

[3] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[4] Th. Fraichard and C. Laugier. On-line reactive planning for a non-holonomic mobile in a dynamic world. In *IEEE Int. Conf. Robotics & Automation*, pages 432–437, Sacramento, CA, Apr. 1991.

[5] E. Gat. Alfa: A language for programming reactive robotic control systems. In *IEEE Int. Conf. Robotics & Automation*, pages 1116–1121, Sacramento, CA, Apr. 1991.

[6] D.M. Lyons. A Process-Based Approach to Task-Plan Representation. In *IEEE Int. Conf. Rob. & Aut.*, Cincinatti, Ohio, May 1990.

[7] D.M. Lyons and A.J. Hendriks. Implementing an integrated approach to planning and reaction. In *SPIE Conference on Intelligent Robotic Systems; Algorithms and techniques*, Boston, MA, November 1991.

[8] D.M. Lyons and A.J. Hendriks. Reactive planning. In S.Shapiro Editor in chief, editor, *Encyclopedia of Artificial Intelligence (2nd Edition)*. John Wiley and Sons, 1992. Also: Philips TR-91-016 (public).

[9] D.M. Lyons, A.J. Hendriks, and S. Mehta. Achieving robustness by casting planning as adaptation of a reactive system. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1991.

[10] R. Pelavin. *A Formal Approach to Planning With Concurrent Actions and External Events*. PhD thesis, Univ. of Rochester, Dept. of Computer SC., Rochester, NY, 1988.

[11] M.J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *IJCAI-87*, pages 1039–1046, 1987.

[12] X. Xiaodong and G.A. Bekey. Sroma: An adaptive scheduler for robotic assembly systems. In *IEEE Int. Conf. Robotics & Automation*, pages 1282–1287, Philadelphia, PA, 1988.