Implementing an Integrated Approach to Planning and Reaction.

D. M. Lyons and A. J. Hendriks

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

Abstract

The ability to act flexibly in an uncertain and dynamic environment is one of the key objectives of robotics. In previous work, we described an approach to this problem which we called the *planner-reactor* approach. This paper reviews that approach and presents our current implementation in depth.

1 Introduction

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. An example of such a robot system is a *kitting robot*: a robot that assembles parts kits which are then fed to hard automation for assembly. The process of assembling the tray requires being able to operate in the face of uncertainty about parts arrival time, quality, changing management objectives, etc, all of which which would hamper hard automation.

It has proven difficult to integrate planning and reaction [10]. We have outlined a solution to this problem[11] which adopts the paradigm of casting planning *not* as the *a priori* generation of actions to be fed to an executer, but rather as the on-going adaptation of a reactive system. We call this the planner-reactor approach. This paper reviews that approach and describes our current progress in implementing a planner-reactor system.

We begin by reviewing our problem domain in Section 2. This is followed by a discussion of the current literature, motivating the creation of the planner-reactor paradigm. Section 4 summarizes that paradigm. Sections 5 and 6 describe the implementation environment and a detailed kitting example carried out with that environment, respectively. We conclude by discussing where this implementation lags of the theory, and where the theory (as yet) falls short of our objectives.

2 Problem Domain: Kitting

A kitting robot is a robot system that puts together assembly kits. Simpler and cheaper automation can construct the assemblies once they have been placed in the kits and routed appropriately. Instead of building a factory full of expensive, intelligent robots, the intelligence and cost are focused into a small number of kitting robots which feed the rest of the factory. However, to achieve this ideal, the kitting robot has to 'iron out' all the uncertainties associated with the assembly process so that it can be done by simpler automation.

The kitting robot has to deal with the following sources of uncertainty: variable arrival rates and poses of incoming parts; errors in incoming parts; substitutability of incoming parts; mixed batches of kits; variable availability of resources; response to events in the downstream automation; and response to factory management advice. Thus, the key characteristic of the kitting robot is not so much its ability to reason about assembly, but rather its ability to choose timely and effective actions to suit an uncertain and dynamic environment.

In order to deal with events as they happen, the kitting robot needs to be capable of *real-time, reactive response.* On the other hand, in order to deal with changes in the environment (e.g., bursts of errors in parts, or failures of resources) or new goals issued by factory management, the kitting robot needs to be able *to plan ahead 'on the fly'.*

3 Background

Previous approaches have had difficulty with the construction of a robot system that successfully incorporates both planning and reaction [10]. Classical AI planning, as exemplified by STRIPS [6] and upon which many planning systems are still based, is not sufficiently robust in uncertain and dynamic environments[12]. The principle disadvantage of the classical approach is its reliance on a static and agent-centered view of the environment.

Reactive approaches, a more recent development exemplified by Brooks' work[4], operate very robustly in some uncertain and dynamic environments — namely those for which the reactions have been pre-programmed — but lack any ability to plan ahead. Another disadvantage to the reactive approach is that in general such systems have been constructed by hand; their robustness depends on the skill of their programmer.

Other researchers have attempted to combine the advantages of these two approaches. The two principal methods have been as follows:

- Have the planner produce an abstract plan, which is then fed to an executer/scheduler which can modify the plan as necessary to deal with the current environment, e.g. Chochon & Alami[5], Xiaodong & Bekey[14].
- Have a reactive system as the lowest-level of an incremental planner; once a (partial) plan reaches this level of abstraction, it is executed reactively, e.g., Gat[7], Spector & Hendler[13].

The disadvantages with the executer/scheduler method are firstly that it may be infeasible to generate *a priori* and to load complete plans in a dynamic and uncertain environment, and secondly the executor/scheduler can only react once it has been given a plan to modify. The disadvantage with the second approach is that the hierarchical relationship is too rigid a linkage between planning and reaction, and again, restricts the reactivity that can be exhibited.

Depending on the problem domain, these limitations may not be restrictions. However, our application domain demands that we maintain reactivity at all times, since the kitting robot is the only filter for uncertainty. We developed an approach which completely preserves the reactivity of the system, while allowing planning to advise the system when it can. This approach is summarized in the next section.

4 The Planner-Reactor Approach

The usual definition of a planner is a program that produces a sequence of operations that when applied, will take the world from its initial state to a goal state. Typically, this plan is then sent to an executer which carries out each of the actions. In contrast our definition of planning is as follows: a planner is a system that continually modifies a reactive system, the reactor, so that its behavior becomes more goal directed. This definition is most similiar to that of Bresina & Drummond[3], but it differs in the way adaptation and perception are used.

4.1 Overview

The reactor is a system containing a network of sensory-motor 'linkages' which we will call reactions. The reactor has several key properties: Unlike a plan executor, a reactor is always actively inspecting the world, and will act *at any time* should one of its reactions be triggered. To make *real-time* reaction possible, all the 'intelligence' in a reactor is encoded into reactions and cross-connections between them, and annotated with time-constraints. The reactor is initialized with a number of generally useful behaviors already in place.

The planner is completely separate from, and concurrent with, the reactor. Rather than seeing the planner as a higher-level system that loads plans into an executor, we see the planner as a system that interacts with the reactor to produce appropriate behavior in the same way that the reactor interacts with the world (Figure 1). The input to the planner includes: a model of the environment (EM) in which the planner is operating; a description of the reactor (R); and advice (G) from a user about objectives the reactor should achieve (e.g., geometric goals such as kit layouts) and constraints the reactor should obey in its behavior (e.g., batch mix or resource usage constraints).

Figure 1: Planner-Reactor-World System

The planner cycles, continually determining from R if the reactor's behavior in the current environment will indeed conform to the restrictions G. If it determines that the reactor will not obey the restrictions, it makes an incremental change to the reactor configuration to bring the reactor more into line with G. As indicated in Figure 1, there are two connections between planner and reactor: *Perceptions* and *Adaptations.*

Adaptations. The planner alters the structure of the reactor by specifying adaptations. An adaptation is essentially an instruction to delete part of the reactor structure or to add in some extra structure. Each individual adaptation should be quite small in effect and scope; large changes in reactor behavior only come about as the result of iterated adaptation. Large adaptations would be equivalent to downloading 'plans' to the reactor; something we want to avoid.

Perceptions. The knowledge needs of the planner and reactor are almost always different. The reactor uses sensory data to determine whether to fire its reactions. The planner needs data to allow it to predict the future state of the environment. Thus, in addition to being able to adapt the reactor to contain new reactions, the planner needs to be able to insert its own sensory data collection processes. To distinguish between reactor and planner sensory data, we refer to any data collected by the reactor as sensory data, but sensory data collected to be sent to the planner will be referred to as perceptual data. Without any sensory data, the planner is detached from the environment. Forcing the planner to receive all and only all the sensory data the reactor needs would generally result in much too much uninteresting (from the planner's viewpoint) data and, additionally, in possibly missing some data needed only by the planner. The perceptual data needed by the planner will be strongly dependent on current goals, and will most likely change as those goals change. By altering the structure of the reactor, the planner can alter what perceptions it receives.

Lyons, Hendriks & Mehta (1991)[11] introduced a framework for modelling planner reactor interactions in a way amenable to formal analysis. The next section briefly presents the formalism we use to build this framework.

4.2 The $\mathcal{RS}$ Model

The $\mathcal{RS}$ model is a special model of computation developed for the sensory-based robot programming domain. Plans, actions and world models are all represented as hierarchical networks of concurrent processes. A number of mathematical techniques have been developed for the analysis of such networks, and a subset of the model has been implemented as a robot programming language. The following is a very brief overview of the model; for a more detailed motivation and description, see Lyons (1990)[8]. A synopsis of the operators is provided in the Appendix.

An $\mathcal{RS}$ description of a process, or network of processes, is called a *schema.* We shall treat the terms 'process' and 'schema instance' as synonyms. 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 together to form networks using several kinds of *process composition operators.* Network definitions are grounded in a set of atomic, pre-defined, processes. The composition operators are: *sequential* `A;B`, process `B` sequentially follows process `A`; *concurrent* `(A,B)`, `A` and `B` are concurrent processes and the network terminates when *both* have terminated; *conditional* `A:B`, `B` is executed on the condition that `A` terminates successfully; and *disabling* `A#B`, `A` and `B` are concurrent processes and the network terminates when *either* terminate. 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* $\mathtt{A} :: \mathtt{B} = \mathtt{A} : (\mathtt{B}, \mathtt{A} :: \mathtt{B})$.

Concurrent processes can communicate messages to one another via *communication ports.* The concurrent composition operation can be augmented with a port-to-port connection relation c, which indicates which ports are connected together, $(\mathtt{A}, \mathtt{B})^c$.

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, which is defined as follows: We say that process `A` evolves into process `B` under condition Ω if `A` becomes equal to `B` when condition Ω occurs; we write this as $\mathtt{A} \xrightarrow{\Omega} \mathtt{B}$. Evolves is axiomitized from the definitions of the composition operators, e.g., $\mathtt{A}; \mathtt{B} \xrightarrow{\Omega \mathtt{A}} \mathtt{B}$.

4.3 Reactor Structure and Adaptation

The reactor, and the way in which it can be adapted over time, is specified using the $\mathcal{RS}$ model. The general form of the reactor is the following concurrent network of processes:

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

where

1. $\mathtt{P}^i$ is a reaction or a reactor segment (as explained below).
2. $\mathtt{E}^i$ is a guard process whose termination can be commanded by the planner. By the definition of '#', when $\mathtt{E}^i$ terminates $\mathtt{P}^i$ is aborted and hence removed from the reactor structure.
3. `ADDPLAN` is the interface through which the planner can add new guarded reactions. The planner can cause `ADDPLAN` to terminate and return a given value m for k. By the definition of '::', this results in $\mathtt{E}^m \# \mathtt{P}^m$ being added into the reactor structure.

Reactions. Reactions are specified as compositions of sensory and motor processes. For example, a useful reaction for a kitting robot might be to notice anytime some instance of part $p1$ arrives in its buffer area and to acquire that object instance and move it into the workspace. This can be expressed as follows:

$$\mathtt{Locate}_{p1}\langle obj \rangle :; \mathtt{Place}_{obj,dest}$$

where $\mathtt{Locate}_m$ is a basic sensory process that inspects the world for an instance of part m and terminates when it finds one, producing a pointer to the instance in *obj*. `Place` then acquires and moves object instance *obj* to location *dest*. The recurrent composition ':;' forces `Locate` to be *continually* recreated. This expresses the conditional goal that *whenever* an instance of $p1$ appears, an attempt is made to acquire and move that instance.

Plans versus Reactor Segments. 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 need to represent these *reactor segments* in such a way that they can be extended or refined incrementally by the planner (via items 2 and 3 in eqn. (1) above).

Reactor segments are sets of reactions that can be enabled or disabled depending on the sensory/task context. We refer to a description of a sensory/task context as a *situation.* These are somewhat similar to Agre & Chapman's[1] *functional-indexical aspects*: relevant properties of the immediate environment and ongoing task. Reactor segments are represented in the following *situation-labeled* form.

$$\mathtt{P}t = \Big|_{s \in sit(t)} (\ \mathtt{CheckSit}_s : \mathtt{Reaction}_{t,s} \# I_{t,s}\ ) \tag{2}$$

as a concurrent set of actions*, each of which carries its own monitor process `CheckSit` to determine if the world and reactor present an appropriate situation with respect to task t in which the action (sequence) should be carried out. $sit(t)$ is the set of situations for task t for a particular world model. Every segment is 'guarded' by a condition monitor `I` that monitors the applicability conditions for the segment; should the monitor fail (`I` aborts), that segment is disabled, and the planner is notified. This allows the planner to build reactor segments that monitor the validity of the assumptions on which they operate.

Adaptation equations of the reactor. We can write down how the reactor is affected by the changes described in items 2 and 3 above:

$$\begin{array}{cccl} (\,\mathtt{R}\,,\mathtt{E}^k\#\mathtt{P}^k\,) & \xrightarrow{Dk} & \mathtt{R} & \textit{where } Dk \textit{ is } \mathtt{E} \longrightarrow \mathtt{STOP} \\ \mathtt{R} & \xrightarrow{Ak} & (\,\mathtt{R}\,,\mathtt{E}^m\#\mathtt{P}^m\,) & \textit{where } Ak \textit{ is } \mathtt{ADDPLAN}\langle k\rangle \longrightarrow \mathtt{STOP} \end{array} \tag{3}$$

That is Dk can be used to *delete* reactions, and Ak can be used to *add* reactions. Thus, we can express the process of reactor adaptation as follows

$$\mathtt{R} \xrightarrow{\alpha 1} \mathtt{R}^1 \xrightarrow{\alpha 2} \mathtt{R}^2 \xrightarrow{\alpha 3} \ldots \mathtt{R}^n \tag{4}$$

where $\mathtt{R}^i$ is the ith improved reactor, and αi is the combination of Ak and Dk conditions that take $\mathtt{R}^{i-1}$ to $\mathtt{R}^i$. We suggested in [11] that this adaptation sequence is an appropriate formalism with which to study issues of reactor convergence and stability. In the same paper, we also gave a definition of reactor improvement based on the concept of iteratively widening the situations covered by the reactor.

5 Implementation Environment

We have implemented an initial version of a planner-reactor system to conduct kitting robot experiments. Figure 2 shows the basic layout of our implementation.

Hardware. A Puma-560 robot is the basis of the kitting workcell. The original VAL-based controller is used to control the manipulator (albeit in a slightly unusual way, as described in the Robot Server section). The Puma is equipped with a four-fingered, 2-DOF pneumatic gripper. The fingers are grouped into two opposing pairs, and each pair can be controlled in position or gripping-force mode. Each finger has a two-axis force sensor, arranged in such a fashion so that when all four fingers are grasping an object, the forces and torques on that object can be computed.

The workcell has two cameras: A global camera situated above the workcell, whose view covers the entire workspace, but consequentially doesn't yield much detail; and a local camera, embedded in the 'palm' of the gripper, whose view is determined by the position and pose of the gripper, and which can be used to get a close-up view of an object. The remainder of the cell consists of the worksurface and a conveyer belt on which parts can be introduced or removed from the worksurface.

The kit consists of three components of a servo moter: the motor, the external shell, and the cap. The tray has three shaped depressions for these objects. In this example, the parts fit into the depressions with very loose tolerance. The parts are delivered to the cell via the conveyer belt. In the current implementation, the object arrival poses are restricted to lie within a specified subset of all possible poses. (This restriction is temporary). The kitting task consists of taking parts as and whenever they come in, putting them into trays, and shipping the trays out on the conveyer belt.

Robot Server. The VAL controller is connected by an RS232 line to one of a set of M68020 processors on a common VMEBUS. This processor functions as a robot server processor. Interface programs written in VAL allow the processor to issue commands to the robot in two modes. In the first mode, joint or endpoint configurations are transmitted and translated to VAL `move` commands. This mode is used to move the robot coarsely around the workspace. In the second mode, endpoint position/velocity commands are transmitted and used to update the endpoint position/velocity at about 10 to 15 Hz. This mode is used for feedback controlled motion such as visual tracking. This latter mode is implemented using the VAL internal `alter` command. The processor can also

* $\Big|$ is the set notation form of the concurrent composition operator.

Figure 2: The Kitting Robot Implementation.

be used to query the robot position. The `robot server` program runs on the M68020 processor, accepting mode and configuration messages, translating them for the VAL interface, and passing back data to the requester when necessary.

Vision Server. The two cameras are connected to a Philips PAPS industrial vision system. This system is controlled by another of the M68020 processors on the common VMEBUS. The processor functions as a vision server processor. Using this processor, programs can be run on PAPS to carry out various 2D image operations. The `vision server` program runs on this processor and accepts two commands: A `recognize` command causes it 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. If there are, then the template name, the 3-D position and the angle in the worksurface plane are returned for each one. The pose restriction mentioned previously comes about because of the limited set of object models. The second command is the `track` command. This causes the vision server to issue commands to the robot server so as to match the position and velocity of the object currently seen by the local camera.

$\mathcal{RS}$ System. A subset of the $\mathcal{RS}$ model has been implemented as a robot programming language. The `RS-L3` programming environment consists of a YACC/LEX-based parser and a real-time executer. The parser accepts systems of process definition equations in a computer-keyboard version of the $\mathcal{RS}$ syntax. It passes on these parsed definitions, as they are made, to the real-time executer, which executes them as dictated by the model semantics. Thus, all execution is interpretive. The executer implements composition, port-i/o, buffer management, and scheduling. It uses threaded-execution to implement concurrency. The thread scheduler implements an *earliest deadline first* real-time scheduler†. The parser always runs on a SUN and communicates, using UNIX sockets, with a remote executer. The executer can be run on one of the VMEBUS processors or on a SUN. For the example described in the next section, the system was run on a SUN (for memory reasons).

A set of basic sensory and motor schemas has been built to interface to the robot and vision servers. These

†For a description of our progress to date in real-time constraints, see Lyons & Hendriks (1991)[9].

basic schemas run as atomic processes that send command messages to the servers and return results from them. 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. $\texttt{Domove}_p$ is a basic motor process that transmits a `move(p)` command to the robot server, and terminates when that command has been carried out.

Reactor. The reactor is a set of `RS-L3` process definition equations, as described in eqn. (1) in this paper. The `E`, `I` and `ADDPLAN` processes are all basic processes, and all communicate with a remote planner. Communication with the planner is via an ethernet UNIX socket link. The `E` and `ADDPLAN` processes can be forced to terminate via this link, and the `I` and perception processes return their information via this link. The planner can also load new process definitions into the system over this link.

Planner. The planner uses Pelavin's[12] Interval Temporal Logic (ITL) formalism to represent world models. It maintains these models as sets of temporal predicates and rules. Temporal relations between intervals of time are maintained in a network and new relations are enforced by constraint propagation [2]. 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).

Actions have executability condition and effect formulas. Once the executability conditions are satisfied, execution of the action will guarantee that the effects will occur in the world. The use of ITL as the basis for our planner enables us to specify and reason about communicating concurrent actions, since executability conditions can include conditions that must hold *while* the action is executing. The planner implements the planning control strategy outlined in this paper and maintains the link with the reactor using asynchronous signal handling to catch incoming perceptions.

6 Kitting Example

Consider the following kitting robot example: The robot looks for a kit tray t, fills in parts[‡] $p1$ and $p2$ (which arrive via a conveyer belt) on that tray, and then sends the tray out of the cell. We will use this example to illustrate how the planner determines what adaptations to make to the reactor in the case i) that it is constructing a reactor segment (a 'plan') from scratch (unusual), and ii) that it is adapting a reactor to suit new goals (more usual).

New Goals. Using the ITL notation, the kitting goal is

$$\begin{aligned} \exists x,y,z,i1,i2[\quad & (Instance(p1,x) \wedge Instance(p2,y) \wedge Instance(t,z))@i1 \\ & \Rightarrow \\ & (IN(x,z) \wedge IN(y,z) \wedge AT(z, outbuffer))@i2 \wedge ITL(i2, finishes, time)\quad] \end{aligned} \tag{5}$$

This conditional goal specifies that when there are instances of $p1$, $p2$ and the tray t available, put the kit together, and place it in the workcell's output buffer *outbuffer*. 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. In trying to establish an appropriate reactor structure, the planner first determines the satisfiability of the antecedent of the goal. Each *Instance* goal can be achieved by an appropriate `Locate` action. In a similar fashion, the consequent of the conditional goal is planned for, and the appropriate actions are, respectively, `Place,` for placing the tray and `Insert,` for inserting the components into the tray. Finally, actions are tagged with the conditions under which they should be executed in the task. Thus the final reactor segment produced is the following (in the situation-labeled form):

$$\begin{array}{lcll} \texttt{Rsegment} & = & (\ \texttt{NoTray} & : \texttt{Locate}_t\langle p\rangle : \texttt{Place}_{p,ws}\ , \\ & & \ \ \texttt{Tray}\langle t\rangle & : (\ \texttt{Locate}_{p1}\langle p\rangle : \texttt{Insert}_{p,t}\ , \\ & & & \quad \texttt{Locate}_{p2}\langle p\rangle : \texttt{Insert}_{p,t}\ ) \\ & & \ \ \texttt{FullTray}\langle t\rangle & : \texttt{Place}_{t,outbuffer}\ ) \end{array} \tag{6}$$

‡Just dealing with two of the three parts, for simplicity.

where ws is the kitting location within the workspace. This can be added using ADDPLAN to the reactor to incrementally produce the following reactor:

$$
\begin{array}{lcl}
\texttt{REACTOR} & = & (\mathop{\Big|}\limits_{i \in 1 \ldots 4} \texttt{E}^i \# \texttt{P}^i \ , \ \texttt{ADDPLAN}\langle k \rangle :: (\texttt{E}^k \# \texttt{P}^k) \ ) \\
\texttt{P}^1 & = & (\texttt{NoTray} \ : \ \texttt{Locate}_t\langle p \rangle : \texttt{Place}_{p,ws}) \ \# \ \texttt{I}^1 \\
\texttt{P}^2 & = & (\texttt{Tray}\langle t \rangle \ : \ \texttt{Locate}_{p1}\langle p \rangle : \texttt{Insert}_{p,t}) \ \# \ \texttt{I}^2 \\
\texttt{P}^3 & = & (\texttt{Tray}\langle t \rangle \ : \ \texttt{Locate}_{p2}\langle p \rangle : \texttt{Insert}_{p,t}) \ \# \ \texttt{I}^3 \\
\texttt{P}^4 & = & (\texttt{FullTray}\langle t \rangle \ : \ \texttt{Place}_{t,outbuffer}) \ \# \ \texttt{I}^4
\end{array}
\tag{7}
$$

where the $\texttt{I}^i$ processes protect the assumptions under which the segment was generated.

Change of Goal. Now, consider the case when the planner receives a new goal to use $p3$ in place of $p2$. Given this modification on existing advice, e.g. $Modify(Goal, p2 \rightarrow p3)$, where $Goal$ is the previous goal given to the planner, the planner first determines which parts of the reactor are affected, by looking at which parts of the reactor achieved goals with $p2$. In this case this is $\texttt{P}^3$:

$$\texttt{P}^3 = \texttt{Tray}\langle t \rangle \ : \ \texttt{Locate}_{p2}\langle p \rangle : \texttt{Insert}_{p,t} \ \# \ \texttt{I}^3 \tag{8}$$

This part of the reactor is not appropriate anymore and has to be replaced. The planner traces back the deduction of this part of the reactor, and determines what modifications and/or additional actions are necessary to achieve the modified goal. In this case, the new reactor part $\texttt{P}^5$ is very similar to the old one:

$$\texttt{P}^5 = \texttt{Tray}\langle t \rangle \ : \ \texttt{Locate}_{p3}\langle p \rangle : \texttt{Insert}_{p,t} \ \# \ \texttt{I}^5 \tag{9}$$

The substitution of this code into the middle of the kitting reactor segment is accomplished by

1. $\texttt{E}^3$ is terminated by the planner causing $\texttt{P}^3$ to abort.
2. ADDPLAN terminates with $k = 5$, adding the newly-defined $\texttt{E}^5\#\texttt{P}^5$ into the reactor.

7 Discussion

In this paper we have reviewed our approach to integrating planning and reaction. This approach ensures that the reactive component of the system is free to react at any time without unnecessary constraint by the planner, while the planner is still free to improve the reactor when it can. We have presented our implementation environment and described an example of kitting that we have implemented in the planner-reactor framework. This example is the subject on on-going refinement.

The implementation described here is not yet complete in the following areas. On-going adaptation, as described by (4), is not implemented: the planner is restricted to making relatively large and relatively infrequent adaptations. The planner-reactor theory needs to be expanded on this issue. In basic design, the planner-reactor falls into the class of what in adaptive control is called *self-tuning regulators*. $\mathcal{RS}$ provides us with a powerful discrete-event control representation: more concise than automaton or petri-net models, and yet with some mathematical tools (such as the evolves operator) available. Using this, we believe that it is possible to construct a stronger theoretical treatment of the iterative effects of planner adaptation.

References

[1] P. Agre and D. Chapman. Pengi: an implementation of a theory of action. In *Proc. AAAI-87*, pages 268–272, Seattle, WA, Oct. 1987.

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

[3] J. Bresina and M. Drummond. Integrating planning and reaction. In J. Hendler, editor, *AAAI Spring Workshop on Planning in Uncertain, Unpredictable or Changing Environments*, Stanford CA, Mar. 27–29 1990. Systems Research Center, Univ. of Maryland.

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

[5] H. Chochon and R. Alami. Nns, a knowledge-based on-line system for an assembly workcell. In *IEEE Int. Conf. Robotics & Automation*, pages 603–609, San Francisco, CA, 1986.

[6] 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.

[7] E. Gat. Robust low-computation sensor-drive control for task-directed navigation. In *IEEE Int. Conf. Robotics & Automation*, pages 2484–2489, Sacramento, CA, Apr. 1991.

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

[9] D.M. Lyons and A.J. Hendriks. Planning and acting in real-time. In *IMACS World Congress on Computation and Applied Math.*, page 1387, University of Dublin, Dublin Ireland, 1991.

[10] 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).

[11] 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.

[12] 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.

[13] L. Spector and J. Hendler. An abstraction-partitioned model for reactive planning. In *Fifth Rocky Mountain Conference on AI*, Las Cruces, NM, June 1990.

[14] 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.

Appendix: $\mathcal{RS}$ Terminology

- Concurrent Composition: $\mathtt{T} = (\mathtt{P},\mathtt{Q})^c$. T behaves like P and Q in parallel, connected by port-to-port relation c. This can also be written as $\mathtt{T} = \Big|_{i\in I} \mathtt{P}^i$. This operator is *idempotent*. Processes are identical if they are concurrent, identically parameterized, and connected instances of the same schema.
- Sequential Composition: $\mathtt{T} = \mathtt{P};\mathtt{Q}$. The process T behaves like the process P followed by Q.
- Conditional Composition: $\mathtt{T} = \mathtt{P}\langle v\rangle : \mathtt{Q}_v$. T behaves like the process P until that terminates. If P aborts, then T aborts. If P stops, then the value v produced by P is acquired, and T then behaves like $\mathtt{Q}_v$.
- Disabling Composition: $\mathtt{T} = \mathtt{P}\#\mathtt{Q}$. T behaves like the concurrent composition of P and Q until either terminates, then the other is aborted.
- Synchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v\rangle :: \mathtt{Q}_v = \mathtt{P} : \mathtt{Q}; \mathtt{P} :; \mathtt{Q}$.
- Asynchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v\rangle :: \mathtt{Q}_v = \mathtt{P} : (\mathtt{Q}, \mathtt{P} :: \mathtt{Q})$.

Table of $\mathcal{RS}$ Composition Operations.

- $\mathtt{P} \xrightarrow{\Omega\mathtt{P}} \mathtt{STOP}$ (definition of termination condition).
- Sequential: $\mathtt{P};\mathtt{Q} \xrightarrow{\Omega\mathtt{P}} \mathtt{Q}$.
- Conditional:
  $\mathtt{P}\langle i\rangle : \mathtt{Q_i} \xrightarrow{\Omega\mathtt{P}} \mathtt{Q_i}$, and also
  $\mathtt{P}\langle i\rangle : \mathtt{Q_i} \longrightarrow \mathtt{ABORT}$ if $\mathtt{P}\langle i\rangle \longrightarrow \mathtt{ABORT}$.
  Different behavior on termination versus abortion is what makes this operator ‘conditional.’
- Network: $(\mathtt{P}^0,\ldots,\mathtt{P^i},\ldots,\mathtt{P^n}) \longrightarrow (\mathtt{P^i},\ldots,\mathtt{P^n})$ iff $\pi(\Omega\mathtt{P}^0,\ldots,\Omega\mathtt{P^{i-1}})$, where $\pi(A,\ldots,B)$ is the disjunction of all orderings (including conjunction) of the conditions $A,\ldots,B$.
- Disabling: $(\mathtt{P}^0\#\ldots\#\mathtt{P^i}\#\ldots\#\mathtt{P^n}) \longrightarrow \mathtt{STOP}$ iff $\Omega\mathtt{P^i}$ for an $i \in \{1\ldots n\}$, in which case for all $j \neq i$ $\mathtt{P^j} \longrightarrow \mathtt{ABORT}$.

STOP is a process that immediately terminates. ABORT is a process that immediately aborts.

Definition of the $\mathcal{RS}$ Evolution Operator.