Planning as Incremental Adaptation of a Reactive System

D. M. Lyons and A. J. Hendriks

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

Abstract

The importance of solving the problem of integrating deliberative ("planning") capabilities and reactive capabilities when building robust, 'real-world' robot systems is becoming widely accepted [4, 12, 22]. This paper presents a solution to this problem: cast planning as the incremental *adaptation* of a reactive system to suit changes in goals or the environment. Our application domain is a manufacturing problem — robotic kitting. This paper represents an advance on existing work in two ways: It presents and formally examines an architecture that incorporates the benefits of a deliberative component *without* compromising the reactive component. Secondly, it provides the first set of performance statistics in the literature for this class of system.

In our approach, the reactive system (the reactor) is a real-time system that continually interacts with the environment, and the planner is a separate and concurrent system that incrementally 'tunes' the behavior of the reactor to ensure that goals are achieved. We call this the *planner-reactor* approach. The reactor is described using a formal framework for representing flexible robot plans, the $\mathcal{RS}$ model [17, 19]. Thus, the behavior of the reactor, and the rules by which the reactor can be modified, become open to mathematical analysis. We employ this to determine the constraints the planner must abide by to make *safe* adaptations and to ensure that incremental adaptations *converge* to a desired reactor. We discuss our current implementation of planner and reactor, work through an example from the kitting robot application, and present implementation results.

Keywords: Planning, Reacting, Adaptation, Robots, Robot Planning, $\mathcal{RS}$, Integrating planning and reaction.

Paper 93A/11, revised version, for publication in *Robotics & Autonomous Systems, V14 N4, 1995*

Planning as Incremental Adaptation of a Reactive System

D. M. Lyons and A. J. Hendriks

Philips Laboratories
Philips Electronics North America Corporation
Briarcliff Manor NY 10510
`dml@philabs.philips.com`

1 Introduction

The importance of integrating deliberative ("planning") capabilities and reactive capabilities when building robust, 'real-world' robot systems is becoming widely accepted [4, 12, 22]. The problem with this integration is that much of the power and robustness of a reactive system comes from its lack of deliberation [6]. This paper presents a solution to this problem: cast planning as the incremental *adaptation* of a reactive system to suit changes in goals or the environment. Our application domain is a manufacturing problem — robotic kitting. This research represents an advance on existing work in two ways: It presents and formally examines an architecture that incorporates the benefits of a deliberative component *without* compromising the reactive component. Secondly, it provides the first set of performance statistics in the literature for this class of system.

Classical AI planning, as exemplified by STRIPS [9] is not sufficiently robust in uncertain and dynamic environments[23]. Reactive approaches, exemplified by Brooks' work[6], were developed in response to this problem. They operate very robustly in some uncertain and dynamic environments — namely those for which the reactions have been pre-programmed. The limitations of purely reactive systems have led, in turn, to the development of so-called *hybrid* systems, systems that integrate planning and reaction, e.g., Robo-Soar [16], Universal Plans [24], AuRA [3], ERE [4], SSS [7], APE [27] and XFORM [20].

Our solution to the kitting robot problem improves on the state-of-the-art in hybrid systems by combining a planner and a reactive system in such a way that the planner can iteratively improve the reactive system to contain novel, autogenerated behaviors, while minimally interfering with the ability of the reactive system to react at *all* times to the environment. Although we have used kitting as a driving application for this work, our solution to the integration approach is general. This approach is equally applicable to other domains that demand the integration of deliberative and reactive capabilities, such as mobile robots, autonomous explorers, emergency response planning, and so forth.

In our approach, the reactive system (the reactor) is a real-time system that continually interacts with the environment, and the planner is a separate and concurrent system that incrementally 'tunes' the behavior of the reactor to ensure that goals are achieved. We call this the *planner-reactor* approach. The reactor is described using a formal framework for representing flexible robot plans, the $\mathcal{RS}$ model [17, 19]. Thus, the behavior of the reactor, and the rules by which the reactor can be modified, become open to mathematical analysis. We employ this to determine the constraints the planner must abide by to make *safe* adaptations and to ensure that incremental adaptations *converge* to a desired reactor. We discuss our current implementation of planner and reactor, work through an example from the kitting robot application, and present implementation results.

The remainder of the paper is laid out as follows: Section 2 introduces robot kitting and motivates it as an application domain for integrating reaction and deliberation. Section 3 first reviews existing work in this area, to illustrate the context for this paper, and then paints the broad picture of the 'planning as adaptation' concept that will be elaborated in subsequent sections. Section 4 provides some background information on our action representation, the $\mathcal{RS}$ model. Section 5 presents the formalization of the reactor and safe adaptation, and section 6 addresses the issue of reactor convergence. Section 7 describes the planner and its process of iterative adaptation. In section 8 we present a detailed example from our planner-reactor implementation of a robotic kitting workcell. In section 9 we present performance results from this implementation.

2 The Kitting Robot Application

In this section, the kitting robot application domain is described, with special attention to the characteristics that demand integration of reaction with deliberation. Kitting is an important domain because it is one of the very few well-defined industrial domains in which reactive systems, and integrated reactive and planning systems, can be shown to be necessary and worthwhile.

A kitting robot is a robot system that accepts the raw materials for a product and places them into 'kits' — trays containing all the components needed to built a particular product. Simpler and cheaper automation can construct the assemblies once they have been placed in the kits and routed appropriately. Kitting provides the line stock availability, the line scheduling flexibility, and the reduced station cycle time productivity for competitive results [26]. 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. Instead of building a factory full of expensive, intelligent robots, the intelligence and cost is focused into a small number of kitting robots which feed the rest of the factory.

The kitting robot has to deal with the following sources of uncertainty and dynamic events:

1. Variable availability and arrival rates and poses of incoming parts.
2. Variable quality in incoming parts. Faulty parts need to be removed before they either cause errors in downstream automation, or become assembled into (faulty or low-quality) products.
3. Mixed batches of kits. The robot may be serving more than one line, or the lines may be running mixed batches.
4. Variable availability of resources. Examples of resources are tools or machines with which the robot needs to coordinate directly.
5. Response to events in the downstream automation. For example, machines breaking down, or alternate machine configurations.
6. Response to factory management advice. For example, response to changing the batch mix or throughput, or implementing once-off fixes for special situations.

To deal with the short term issues in filling assembly kits, e.g., selecting and executing actions to acquire product parts, the kitting robot needs to be capable of *real-time, reactive response*. To handle longer-term concerns such as changes in factory management instructions (new goals) or changes in the operating conditions (changing environment), the kitting robot needs a *deliberative* or planning component.

Some aspects of planning for kitting, such as kit assembly orders, can be done off-line, in the same fashion that assembly planning is done. Other aspects of the planning problem, nonetheless, need to be handled on-line. Factory management instructions such as new batch mixes or parts substitutions cannot be entirely planned off-line because the information about what kits are being mixed or what parts are substitutable is simply not available. Similarly, changes in the operating environment such as bursts of errors in parts or failures of resources cannot be entirely anticipated off-line due to lack of specific information. For these reasons, the kitting robot needs a certain amount of on-line planning capability. The problem of building a robot control system that successfully incorporates both reaction and on-line planning is still under active exploration by both the Robotics and AI communities.

3 The Planner-Reactor Approach

This section begins with an overview of current work in the area of integrating reaction and deliberation. An informal introduction to our approach to integrating reaction and deliberation is then presented. Additional detail and a formal treatment will be presented in subsequent sections.

3.1 Review of Existing Work

Purely deliberative approaches (e.g., [9]) are good at determining globally good courses of action for a robot, selecting an optimal assembly sequence for example. They are not very robust, however, in the face of changing or uncertain conditions in the robot's environment. Typically, a deliberative system expends a lot of effort deriving a good course of action for the robot, only to find on starting that the environment has changed and rendered the actions unsuitable [22]. Purely reactive approaches (e.g., [6, 13]) operate very robustly in some uncertain and dynamic environments — namely those for which the reactions have been manually pre-programmed. However, many application domains such as mobile robots[12], autonomous explorers[4], emergency response planning [22] as well as our domain, robotic kitting, demand the integration of deliberative and reactive capabilities.

One approach to allowing both on-line planning and also action is *interleaved planning and execution* (e.g., Robo-Soar [16]) — plan a certain amount, then execute these actions, then plan some again, and so on. This approach requires *one* of either planning *or* execution to be taking place at any time. This causes problems if the system is in a planning phase when an unexpected event occurs – planning interferes with reaction to uncertain and dynamic events.

Schoppers[24] amongst others has proposed an 'off-line' generator for reactive systems; Xiaodong and Bekey [28] also suggest something similar for an assembly cell equipped with a reactive scheduler. This approach works well when the deliberative component can be separated 'off-line.' We address the problems that arise when both reactive and deliberative components need to be 'on-line.'

Connell's SSS [7] and Arkin's AuRA [3] addresses this integration for mobile robots. In SSS the deliberative and reactive components are asynchronously linked, allowing reaction and planning to occur simultaneously and with minimal interference. This minimal interference approach is also necessary for the kitting domain. SSS, however, is restricted to enabling/disabling behaviors, while we need to be able also to autogenerate completely *new* behaviors.

AuRA also allows for asynchronous interactions, and shares with $\mathcal{RS}$ an approach based on schemas. AuRA employs a plan-then-execute paradigm, where a reactive plan is first produced in complete form, and then loaded into the executor. In the case of highly conditional plans, such as we have for kitting, this may result in unrealistically long planning delays. Thus, we would prefer to adopt an incremental adaptation approach to the planner's update of the reactive system. In this case, the adaptation problem is primarily one of *adapting structure* as opposed to adapting parameters – adding in novel parts to, or removing parts from, an action strategy.

Bresina & Drummond's ERE [4] and McDermott's XFORM [20] come closest to our approach: a planning system that incrementally improves an asynchronously connected reactive system. A key difference is that the kitting domain has a strong *repetitive* activity element, whereas ERE and XFORM are oriented to primarily once-off activities. This repetitive element can be exploited to define a notion of iterative improvement that is not available in a once-off activity domain.

For the overall system to retain a reactive quality, it is necessary that the deliberative and reactive components of the systems be separate and function asynchronously. If they were not asynchronous, the reactive component might need to wait for plans/instructions from the deliberative component, or wait for acknowledgement before continuing with actions (as in e.g., a hierarchical scheme). The asynchronous relationship is *key* to preserving reactive capabilities while, nonetheless, allowing for improvement of performance by the addition of a deliberative viewpoint.

3.2 The Planner-Reactor Architecture

In our approach, a planner is a system that continually modifies a concurrent and separate reactive system so that its behavior becomes more goal directed. We refer to the reactive system as the *reactor* after the terminology established by Bresina & Drummond. Figure 1 illustrates the architecture.

<u>Reactor.</u> The reactor contains a network of reactions — hardwired compositions of sensory and motor actions. The key property of the reactor is that it can produce action at *any* time. Unlike a plan executor,

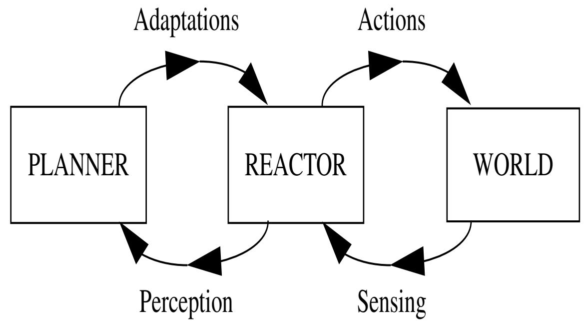

Figure 1: The Planner-Reactor-World System.

a reactor can act independently of the planner; it is always actively inspecting the world, and will act should one of its reactions be triggered. A reactor should produce timely, useful behavior even *without* a planner.

Planner. 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 an equal-level system that continually *tunes* the reactor to produce appropriate behavior. The interaction between the planner and reactor is entirely *asynchronous*. The input to the planner includes: a model of the environment in which the planner is operating; a description of the reactor's structure; and information from the user about objectives the reactor should achieve (e.g., in the kitting problem domain, geometric goals such as kit layouts) and constraints the reactor should obey in its behavior (e.g., batch mix or resource usage constraints). The planner continually determines if the reactor's responses to the current environment would indeed conform to the objectives. If not, then the planner makes an incremental change to the reactor configuration to bring the reactor's behavior more into line with the objectives.

Planner-Reactor Interactions. As indicated in Figure 1, there are two routes of interaction between planner and reactor.

1. Adaptations. The planner alters the structure of the reactor by specifying adaptations.
2. Perceptions. These are sensory data collected by the reactor to be sent to the planner.

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 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 seek to avoid.

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 sensory data to allow it to predict the future progress of the environment and the status of the reactor.

Before we present the planner-reactor approach in more detail, it is necessary to describe the language we use to build and analyze the reactor.

4 $\mathcal{RS}$

In [19] a special purpose model of computation for robot programming was developed. That work began by listing the computational characteristics of robotics, including the importance of parallelism, the fundamental nature of sensing and action, and of the sensing-action 'loop.' The model was called *Robot Schemas*; $\mathcal{RS}$ for short. It viewed robot programs as networks of concurrent interacting processes. In later work [17, 18], the model was extended to include a set of process composition operators to simplify the specification and analysis of action ordering.

Much of the model was inspired by Arbib's Schema Theory [2] (an approach to representation incorporating both AI and Brain Theory perspectives). It was also strongly influenced by work in cooperating sequential processes [15], and it belongs to a body of work on using that computational paradigm to model and analyze real-time systems, e.g., LOTOS [5], COSPAN [14].

We will employ $\mathcal{RS}$ to represent the reactor and the mechanism by which it is improved by the planner. In addition to being a formal model, $\mathcal{RS}$ has also been implemented as a robot programming language. There are other existing reactive robot programming languages, e.g., Gat's ALPHA [13], Firby's RAPS [10] and McDermott's RPL [21] amongst others. The single crucial difference between $\mathcal{RS}$ and these languages is the $\mathcal{RS}$ emphasis on being able to analyze behavior formally. A second difference, introduced in [18], is the use of $\mathcal{RS}$ to model environments as well as controllers.

4.1 Processes

An $\mathcal{RS}$ description of a process, or network of processes, is called a *schema*. For example, $\mathtt{P}_m\langle r\rangle$ denotes a process that is an instance of the schema `P` with one ingoing parameter m and one outcoming result r (see figure 2). Networks are built by composing processes together using several kinds of *process composition operators*. This allows processes to be ordered in various ways, including concurrent, conditional and iterative orderings. At the bottom of this hierarchy, every network must be composed from a set of atomic, pre-defined processes. The set of *basic schemas* defines what processes are atomic.

4.2 Compositions

The $\mathcal{RS}$ composition operations are *sequential* `A;B`, *concurrent* `A | B`, *conditional* `A:B` , *negation* `~A` and *disabling* `A#B`. We employ two non-atomic operators, *synchronous recurrent* `A:;B`, and *asynchronous recurrent* `A::B`, and we define these below.

In *sequential composition,* the process `T = P;Q` is the network of the process `P` executed first, and when that terminates, process `Q` executed until it terminates. This is used to enforce a strict ordering on operations, e.g., $\mathtt{Place}_{part1};\mathtt{Place}_{part2}$.

Concurrent composition indicates that two or more processes should be carried out concurrently. This allows us to represent a lack of ordering between activities, e.g., ($\mathtt{Place}_{part1} \mid \mathtt{Place}_{part2}$), or parallel actions — actions which need to be done simultaneously, e.g., squeezing an object *obj* with two fingers $f1$ and $f2$: ($\mathtt{ApplyForce}_{f1,obj} \mid \mathtt{ApplyForce}_{f2,obj}$). Concurrent processes can communicate messages to each other via *communication ports.*

Conditional composition allows the construction of networks whose behavior is conditional. The network of `T = P:Q` behaves like `P;Q` iff `P` terminates successfully. If `P` aborts, then `Q` is not carried out, and `T` aborts. For example, in $\mathtt{LidOpen}_{box} : \mathtt{Place}_{x,box}$ whether `Place` is carried out or not depends on whether `LidOpen` terminates successfully or not. A wide range of traditional CASE and IF forms can be built from this operator.

Negation composition simply inverts the termination condition of a process. If `A` terminates successfully, then `~ A` terminates unsuccessfully, and vice-versa.

Disabling composition allows one process to terminate another. The network `T = A#B` behaves like (`A | B`) except that it terminates whenever either process terminates.

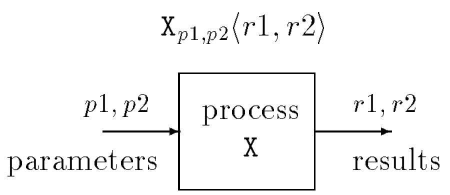

Figure 2: Notation for Process Parameters and Results

None of the composition operators introduced so far allow us to have repeated actions. We will make use of some special forms of recursion to achieve repetition. The schema $\mathtt{T}_a = \mathtt{X}_a : \mathtt{T}_{f(a)}$ is an example of a guarded recursive process definition. We call it guarded because the conditional composition ensures that the recursion can be terminated by X aborting. $\mathtt{X}_a$ must here be a process that evaluates some condition on its argument and terminates successfully if the condition is true, but aborts if the condition is false. An example is the basic process $\mathtt{GTR}_{a,b}$ which succeeds if $a > b$ but aborts otherwise.

The final two composition operators are defined recursively in terms of the first four. In programming terms, one could think of them as 'macros'. *Synchronous recurrent* composition is defined: $\mathtt{A} :; \mathtt{B} = \mathtt{A} : (\mathtt{B}; (\mathtt{A} :; \mathtt{B}))$. The behavior of this composition is similar to *while-loop* iteration with A as the test and B as the body of the loop.

Asynchronous recurrent composition is defined as $\mathtt{A} :: \mathtt{B} = \mathtt{A} : (\mathtt{B} \mid (\mathtt{A} :: \mathtt{B}))$. This composition does not iterate, but rather is a loop that 'spawns' off its body B every time its 'condition' A is satisfied.

4.3 If-Then-Else

We employ the following concise If-Then-Else form in the later parts of this paper and we therefore introduce it here.

$$\sim (\ \mathtt{COND}\ :\ \mathtt{THENPART}\ )\ :\ \mathtt{ELSEPART} \tag{1}$$

If the process COND terminates successfully, then by the definition of conditional composition, THENPART is next executed, and if it terminates successfully, then this innermost conditional composition terminates successfully. The negation composition, however, inverts the termination condition. Thus, again by the definition of conditional composition, ELSEPART is not executed, and the outermost conditional composition terminates unsuccessfully. On the other hand, if COND terminates unsuccessfully, then the innermost conditional composition terminates unsuccessfully, but because of the negation, the outermost one terminates successfully and ELSEPART is executed.

4.4 Process Evolution

The $\mathcal{RS}$ language as introduced to this point is sufficient for the purposes of building programs. However, to analyze how such programs would behave, it is necessary to be able to examine how process networks evolve over time, as processes dynamically terminate or are created. For this purpose the *evolves* operator was introduced in [17, 18]. This operator 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 $\mathtt{A} \xrightarrow{\Omega} \mathtt{B}$. A process may have the potential to evolve to a number of different networks. For example the network $\mathtt{A} \mid \mathtt{B} \mid \mathtt{C}$ could evolve to $\mathtt{A} \mid \mathtt{B}$ if C terminates first, or $\mathtt{A} \mid \mathtt{C}$ if B terminates first, and so on. A process must evolve to precisely one of the networks to which it can potentially evolve.

Evolves can be intuitively understood as a "simulated execution" of the network. More formally, evolves is axiomatized from the definitions of the composition operators, e.g., $\mathtt{A};\mathtt{B} \xrightarrow{\Omega\mathtt{A}} \mathtt{B}$. To apply the evolves operator it is necessary to know the successful and unsuccessful termination conditions for each basic process. Evolves is not part of the robot programming implementation of $\mathcal{RS}$. Rather, this operator will be our main mathematical tool in exploring how the reactor behaves under repeated improvement by the planner. (See [18] for a complete description of evolves.)

5 The Structure of the Reactor

Recall from section 3 that the reactor is an independent, reactive system whose structure is such that it can be incrementally adapted by the planner. We begin the description of the reactor by introducing a concept called *reactor situations.* This concept will allow us to build modular and efficient reactors. We then present the fundamental description of the reactor as an $\mathcal{RS}$ process network. We show how this description allows the reactor structure to be adapted incrementally. There are two important issues that arise when adapting a reactor: First, what are the constraints on making adaptations to a reactor *while it is operating* in a 'safe' way (and we shall define 'safe' more precisely). Second, what are the constraints on incrementally adapting the reactor so that it will converge on the 'best' structure (and, again, we will define 'best' more precisely). The formal aspect of $\mathcal{RS}$ allows us to address these issues in a precise fashion.

5.1 Reactor Situations

Reactor situations are a mechanism to group related reactions together in a hierarchical fashion. A reaction is a composition of a sensory process and a motor or action process. This modularity is important from three perspectives:

1. Design: It allows (human) designers to more easily understand the reactor and diagnose any problems that may occur.

2. Adaptation: It provides the planner with a unit around which to define safe changes to the reactor structure.

3. Efficiency: It gives us a way to give computational resources to those reactions that currently need it, while 'suspending' those reactions that are not currently needed.

Intuitively: a situation being active expresses the appropriateness for the reactor to undertake the set of reactions associated with that situation. For example, let `FillTray` situation contain a set of concurrent reactions which together place the parts into the kit. When the situation `FillTray` is asserted, these reactions are enabled and begin to execute. How and when these reactions produce actions, and thus how the kit is populated, will be governed by the sensing process associated with each reaction. (When the sensing process terminates and initiates the action process, we refer to this as the reaction 'firing'.) One of the reactions in the situation should fire only when all the parts are in the tray, and its action should be to terminate the situation and hence disable the kit assembly reactions.

The `FillTray` situation might be asserted as the response action of yet another reaction, one which fires, for example, whenever a kitting tray enters the workspace. In this way, situations can be nested hierarchically.

Many situations may be active at one time and thus the execution of their reactions will be intermingled. If the kit that `FillTray` puts together consists of three parts in a kitting tray, then `FillTray` might consist of one reaction that activates three instances of the `LacksPart` situation, each parametrized with one of the three part types. When a situation is given parameters, such as the part type in this case, this means that all the reactions in that situation can access the value of those parameters. The reactions in `LacksPart` would first search for and then place into the kitting tray a part of the type given by the situation parameter. Activating three instance of `LacksPart` in parallel would mean that the searching for and placing of the three parts are interleaved as opportunities for each are provided in the environment (cf., opportunistic scheduling [11]).

5.2 Representing Reactions and Situations

Reactions are hard-wired responses to sensory conditions. They are represented in $\mathcal{RS}$ as compositions of sensory and action 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:

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

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.

We introduce a small set of basic processes with which to build reactor situations in $\mathcal{RS}$; the set is shown in table 1. This set provides an implementation independent view of situations. An instance of a situation is asserted by the execution of the $\mathtt{ASSERT}_{s,p1,p2,\ldots}$ process, where $p1, p2, \ldots$ are the values

- $\texttt{ASSERT}_{s,p,\ldots}$ Assert situation s and initialize the situation parameters to $p, \ldots$ This terminates when the situation is terminated and stops or aborts depending on whether `FAIL` or `SUCCEED` was used.
- $\texttt{FAIL}_{s,k}$ Terminates instance k of situation s with fail status.
- $\texttt{SUCCEED}_{s,k}$ Terminates instance k of situation s with success status.
- $\texttt{SIT}_s\langle k, p1, p2, \ldots\rangle$ Terminates if an instance of situation s is currently asserted. k is the instance number and $p1, p2, \ldots$ are the current values of the situation parameters.

Table 1: Situation basic processes.

for the parameters associated with the situation. Each situation instance is allocated a unique situation identifier k. A situation instance with unique identifier k is terminated successfully by the execution of the $\texttt{SUCCEED}_{s,k}$ process or unsuccessfully by the $\texttt{FAIL}_{s,k}$ process. The `ASSERT` process terminates only when its situation instance terminates, and it terminates successfully or unsuccessfully (i.e., aborts) depending on whether the situation instance terminated successfully or unsuccessfully.

The $\texttt{SIT}_s\langle k, p1, \ldots\rangle$ process, when executed, suspends itself until an instance of situation s is asserted. It then terminates and passes on the details of the situation instance as its results. The results produced by `SIT` are the unique identifier for this situation instance k and the values of the situation parameters initially set by `ASSERT`.

We use the $\texttt{SIT}_s$ process to ensure that a reaction associated with a situation is only 'enabled' when an instance of that situation is asserted. For example, let $\texttt{P1}, \ldots, \texttt{P}n$ be the reactions for situation s, then we would represent this in the reactor as the following concurrent network:

$$\texttt{PS} = \texttt{SIT}_s :; \texttt{P1} \mid \texttt{SIT}_s :; \texttt{P2} \mid \ldots \mid \texttt{SIT}_s :; \texttt{P}n = \Big|_{i \in 1 \ldots n} \texttt{SIT}_s :; \texttt{P}i \tag{2}$$

The ':;' operation ensures that as long as the situation is active, the reactions are continually re-enabled. Note that once enabled, a reaction cannot be disabled until it has terminated. The asserting and termination of instances of situation s happen as the side effects of $\texttt{ASSERT}_s$ and $\texttt{FAIL}_s$ or $\texttt{SUCCEED}_s$ processes in reactions in other situations.

5.3 Adapting the Reactor

The reactor consists of a network of situation triggered reactions (as described in the previous section) and a well-defined interface through which the planner can modify the reactor structure. The general form of the reactor is the following concurrent network of processes:

$$\texttt{REACTOR} = \texttt{E}_1\#\texttt{SP1} \mid \ldots \mid \texttt{E}_n\#\texttt{SP}n \mid \ \texttt{ADD}\langle k\rangle :: (\texttt{E}_k\#\texttt{SP}k) \tag{3}$$

where

1. $\texttt{SP}i$ is a situation-triggered reaction
2. $\texttt{E}_i$ is a guard process by which (by the #-composition) $\texttt{SP}i$ can be removed by the planner.
3. `ADD` is the interface through which the planner can add (by the ::-composition) new reactor structure in the form $\texttt{E}_m\#\texttt{PS}m$.

The only ways the planner can affect the reactor is by causing an $\texttt{E}_n$ (for specified n) process to terminate, and hence causing some part of the reactor structure to abort, or by causing `ADD` to terminate with a given 'label' m, and hence causing some additional structure to be added into the reactor. Of course, for $\texttt{SP}m$ to be added into the reactor, the process $\texttt{SP}m$ must have been previously defined for $\mathcal{RS}$. So when making adaptations, the planner must first define its new additions as processes, and then instruct ADD to terminate and hence add those new additions into the reactor. Note that if the set of SPm processes is fixed, then this system is very similar if not identical to Connell's approach [7]. However, in our approach the planner can define *completely new* SPm processes at any point and use the mechanism above to enter them into the reactor. The examples in section 8 will make this clear. This novelty is important: it is a lot easier to add a new kit model or object to the planner than it is to write all the possible SPm processes that might be necessary to handle the new kit model or object.

Strictly limiting the ways in which the planner can affect the reactor is important in formalizing the process of adaptation. It is possible to envision versions of our architecture where the planner and reactor might exist on the same processor, even as parts of the same piece of computer code. Even in this extreme case, there would still be a logical division between the "planner," as the part that does the adapting, and the the "reactor," as the part that is adapted. Whether the adaptation to the SPi processes is internally (in the extreme case) or externally (in the normal case) triggered, the crucial thing is to capture how changes can occur.

We can write down how the reactor is affected by the changes described in items 2 and 3 in our reactor definition (3):

$$\begin{array}{ccc} \mathtt{R} \mid (\mathtt{E}^k \# \mathtt{SP}k) & \xrightarrow{Dk} & \mathtt{R} \\ \mathtt{R} & \xrightarrow{Ak} & \mathtt{R} \mid (\mathtt{E}^m \# \mathtt{SP}m) \\ & & \\ \multicolumn{3}{l}{\textit{where } Dk \textit{ is } \mathtt{E}^k \longrightarrow \mathtt{STOP}} \\ \multicolumn{3}{l}{\textit{where } Ak \textit{ is } \mathtt{ADD}\langle k \rangle \longrightarrow \mathtt{STOP} \ \&\ k = m} \end{array} \tag{4}$$

Saying that a process SP successfully terminates (under some condition) is the same as saying that SP evolves to STOP (under that condition). So in (4) above, Dk can be used to *delete* reactions, and Ak can be used to *add* reactions. Using the evolves operation, we can express the process of reactor adaptation as follows:

$$\mathtt{R} \xrightarrow{\alpha} \mathtt{R}' \tag{5}$$

where $\mathtt{R}'$ is the adapted reactor and α is some combination of Ak and Dk conditions.

5.4 Representing Situation Hierarchies

Situations can be nested hierarchically; that is, when situation a asserted, it may in turn cause situation b to be asserted, and a will not finish at least until b has finished. The difficulty in providing hierarchical representation for reactions is that there is a danger of losing the ability to make small, incremental changes to the reactor structure because of the depth of the hierarchy. We avoid this by implementing situation hierarchies on top of a 'flat' set of reactions.

We use the following simple examples to illustrate how hierarchically nested situations are represented and adapted. We consider a reactor R0 that contains a single higher-level situation sa, which in turn contains a single lower level situation sb. We will assume the situation control processes shown in Table 2. This reactor is represented as follows:

$$\begin{array}{rcl} \mathtt{R0} & = & (\mathtt{E}_1 \# \mathtt{P1}) \mid (\mathtt{E}_2 \# \mathtt{P2}) \mid (\mathtt{E}_3 \# \mathtt{P3}) \mid \ \mathtt{ADD}\langle k \rangle :: (\mathtt{E}_k \# \mathtt{P}k) \\ & & \mathtt{P1} = \mathtt{Q} :; \mathtt{ASSERT}_{sa} \\ & & \mathtt{P2} = \mathtt{SIT}_{sa} :; \mathtt{ASSERT}_{sb} \\ & & \mathtt{P3} = \mathtt{SIT}_{sb} :; \mathtt{RSB} \end{array} \tag{6}$$

The reactor is a set of concurrent processes as described in (3): P1, P2 and P3 plus their guard processes $\mathtt{E}_1, \mathtt{E}_2$ and $\mathtt{E}_3$ plus the ADD network. In P1, situation sa is asserted ('triggered') by some process Q which is the 'root' of the situation hierarchy for situation sa. There is one reaction for situation a, P2, the unconditional assertion of situation sb. Situation sb in turn has one reaction in P3, the unconditional activation of some arbitrary reaction RSB, which is the sole 'leaf' in the situation sa hierarchy. Note that even though sa and sb are hierarchically nested, they are *not* represented in this fashion, i.e., nested, in the reactor. If we were to nest them, we would get:

$$\mathtt{R0} = (\mathtt{E}_1 \# (\mathtt{Q} :; \mathtt{ASSERT}_{sa} \mid (\mathtt{E}_2 \# (\mathtt{SIT}_{sa} :; (\mathtt{ASSERT}_{sb} \mid \ldots)))))$$

- $\mathtt{WAIT}_s$ Terminates when no instance of situation s is active.
- $\mathtt{KILL}_n$ Causes the process $\mathtt{E}_n$ to terminate.
- $\mathtt{INHIBIT}_s$ Prevents any reaction of situation s from being enabled.
- $\mathtt{UNINHIBIT}_s$ Removes the inhibition on situation s.
- $\mathtt{WAITINHIBIT}_s$ An atomic combination of WAIT and INHIBIT on situation s.

Table 2: Situation Control basic processes.

In the nested representation, reactions associated with *sb* are not even created until *sb* becomes asserted. Thus, extensive nesting could easily prevent higher-level reactions from responding in a timely fashion to the environment. In addition, the only way to adapt these reactions is to redefine the reaction for situation *sa*. With hierarchical situations, if we nest the reactions, we lose the ability to make local changes to the structure without redefining everything. Therefore, we 'flatten' the hierarchy. The reactions for all situations all start concurrently. Now *any* reaction for any situation can be locally removed without changing anything else.

This flat representation of the hierarchy may seem inefficient since all situations are created at start up time. In practice, this is not a problem, since the SIT process simply puts itself onto a queue awaiting the assertion of its situation. Thus, although all situations are created, the only ones actually running are those that have been asserted.

5.5 Safely Adapting a Hierarchical Reactor

The adaptation equations (4) of the reactor capture structural change in a straightforward manner. Reactions are removed immediately upon the termination of their guard (E) process, and are introduced immediately upon termination of the ADD process. In practice this is not sufficient. It is not always appropriate to interrupt a reaction that is in progress. New reactions may have to be started together, not one by one.

We argue that the following properties are desirable in an adaptation mechanism. When the planner initiates an adaptation event, it must be:

1. *Consistent:* Only old behavior (behavior generated by the old reactor structure) or only new behavior (behavior generated by the new reactor structure) should be produced, not a mix of the two.

2. *Safe:* No reaction should be interrupted by the change.

3. *Bounded:* The adaptation event will terminate and the reactor will produce only new behavior within bounded time.

We now introduce a *general purpose* adaptation mechanism that obeys these properties. We assume the situation control processes shown in Table 2 and introduce the adaptation mechanism via an example.

Consider adapting R0 in the example (6) of the previous section so that situation *sa* is composed of two situations *sx* and *sy* in sequence, each of which have their own reactions RSX and RSY.

$$
\begin{aligned}
\mathtt{R1} \;=\; & (\mathtt{E}_1\#\mathtt{P1}) \mid (\mathtt{E}_4\#\mathtt{P4}) \mid (\mathtt{E}_5\#\mathtt{P5}) \mid (\mathtt{E}_6\#\mathtt{P6}) \mid \; \mathtt{ADD}\langle k\rangle :: (\mathtt{E}_{\mathbf{k}}\#\mathtt{P}k) \\
& \mathtt{P1} = \mathtt{Q} :; \mathtt{ASSERT}_{sa} \\
& \mathtt{P4} = \mathtt{SIT}_{sa} :; (\mathtt{ASSERT}_{sx} ; \mathtt{ASSERT}_{sy}) \\
& \mathtt{P5} = \mathtt{SIT}_{sx} :; \mathtt{RSX} \\
& \mathtt{P6} = \mathtt{SIT}_{sy} :; \mathtt{RSY}
\end{aligned}
$$

An adaptation event is initiated and controlled by the planner using the basic reactor adaptation equations (4). An adaptation event occurs in two phases: first, the deletion of reactor structure occurs *top-down*; second, the addition of new structure occurs *bottom-up.* This separation enforces the consistency property. The first step is to identify the highest-level situation involved in the adaptation; in our example, *sa*. We cannot begin the change while any instance of *sa* is asserted, otherwise one of its reactions might be interrupted (violating safety). By its definition (see table 2) the $\texttt{WAIT}_{sa}$ process will delay until no instance of *sa* is asserted. To guarantee the *boundedness* property it is thus necessary to assume that all situations terminate. Furthermore, the numeric upper bound on adaptation is thus the duration of the longest situation possible.

Once $\texttt{WAIT}_a$ terminates, it is necessary to stop any more instances of *sa* enabling their reactions while the adaptation is in progress. The $\texttt{INHIBIT}_a$ process does this. However, if we do these actions *separately*, then a could become asserted between WAIT terminating and INHIBIT starting, thus causing a violation of our safety constraint. For this reason, we need an atomic composition of these two processes. By its definition (see table 2) the $\texttt{WAITINHIBIT}_a$ process ensures that as soon as no instance of a is asserted, a is immediately inhibited.

We are now in a position to begin 'surgery'. The reactor structure is deleted top-down using the KILL process; in our example, $\texttt{KILL}_2;\texttt{KILL}_3$ will cause the $\texttt{E}_2$ and $\texttt{E}_3$ processes to terminate, excising P2 and P3 from the reactor. The new structure is defined and introduced bottom up. That is, P4, P5 and P6 are defined, and ADD then terminated with $m = 6, 5$ and 4. Each termination introduces *one* new reaction Pm. Note that instances of *sa* may have become asserted while the adaptation was in progress. No *sa* reactions can respond due to the INHIBIT process. When the new additions have all been made, the *sa* situation can be uninhibited with $\texttt{UNINHIBIT}_a$ and all the new reactions become open for use at that point.

We argue that this mechanism is consistent, safe and bounded if the reactor abides by the assumption that *all situations eventually terminate.* Consistency is enforced by the separation of the deletion and addition phases. Safeness is enforced by the WAITINHIBIT process. Boundedness is enforced since the top-down and bottom-up traversals of the situation hierarchy are guaranteed to terminate and provided that the initial WAITINHIBIT eventually terminates. This latter can only occur if all situations eventually terminate.

It is possible to construct algorithms with stronger boundedness. For example, INHIBIT can be used to 'freeze' a running situation. However, these may require the planner to deliberate about cleaning up after the interrupted situation. Requiring all situations to eventually terminate is one way to guarantee that there exists a safe-state of the reactor in which changes can be made. Moreover, this is a local measure of safeness: only the situations to be updated need be checked when a change is to be made.

For the safe adaptation mechanism to guarantee introduction of the updated structure we need to guarantee that every situation will eventually terminate. Thus, we are constrained to write our situation reactions to always terminate. Nonetheless, there are still times when we will need a situation to only terminate successfully when the task succeeds. To handle this we will use a variation on ASSERT that we will call REASSERT defined as follows:

$$\texttt{REASSERT}_{s,p,\ldots} = (\ \sim \texttt{ASSERT}_{s,p,\ldots}) :; \texttt{STOP}$$

The process $\texttt{REASSERT}_s$ will assert situation s and should that situation fail, then it will automatically reassert the situation. The REASSERT process will only terminate when the situation terminates successfully.

6 Improving the Reactor

The planner needs to be able to build a reactor that can operate over the entire range of environmental conditions not just the current conditions. A reactor that fulfills this daunting criterion is called an *ideal reactor* and is similar in behavior to Schoppers' universal plan [25].

6.1 The Ideal Reactor

For a given goal G and environment model EM , we write the the ideal reactor as the $\mathcal{RS}$ process $\texttt{R}^*_{EM,G}$. 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:

$$(\ \texttt{R}^* \ \mid\ \texttt{EM}\ ) \longrightarrow (\ \texttt{R}^* \ \mid\ \texttt{GEM}\ ) \tag{7}$$

That is, when the ideal reactor is run concurrently with a process, `EM`, implementing the environment model the only possible evolutions are those leading to an environment in which the goal is satisfied `GEM`. This is a very general description, but it is sufficient for defining reactor improvement.

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 expect that the planner will always be able to generate the ideal reactor all at once, for the following reasons:

1. It may not have sufficient time because of 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 *ever* build 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.

6.2 Incremental Construction of the Ideal Reactor

It is reasonable to demand that the planner respect the following constraint: Incremental changes to the reactor should result in another working reactor which is at least as capable of achieving its goals under as wide a range of conditions as the original reactor. We therefore introduce a restricted version of the ideal reactor, $\mathtt{R}^{\omega}$

$$(\ \mathtt{R}^{\omega}\ \mid\ \mathtt{EM}\ ) \xrightarrow{\omega} (\ \mathtt{R}^{\omega}\ \mid\ \mathtt{GEM}\ ) \tag{8}$$

This says that as long as assumptions ω hold, then $\mathtt{R}^{\omega}$ behaves like the ideal reactor, $\mathtt{R}^{*}$. In planner terms, ω is a set of assumptions under which `R` has been constructed. Thus there are no "unused" assumptions in ω: the planner does not need to include any assumption it doesn't use. For the remainder of this work we will assume that ω is finite. (To be correct, the ω-ideal reactor for environment `EM` and goals G should be written $\mathtt{R}^{\omega}_{EM,G}$.)

This view of ω suggests the following incremental plan construction strategy: ω is initially chosen to allow the planner to quickly produce a working reactor, and then gradually ω is relaxed over time. We define our assumption relaxation priorities by imposing an ordering '>' on ω based on a function $l()$. We can now rewrite the evolution of the reactor under adaptation by the planner (5) as:

$$\begin{array}{c} \mathtt{R}^{\omega^1} \xrightarrow{\alpha^1} \mathtt{R}^{\omega^2} \xrightarrow{\alpha^2} \mathtt{R}^{\omega^3} \xrightarrow{\alpha^3} \ldots \xrightarrow{\alpha^n} \mathtt{R}^{*} \\ for \quad l(\omega^1) > l(\omega^2) > l(\omega^3) > \ldots > 0 \end{array} \tag{9}$$

where α^i is some combination of Ak and Dk conditions. A good choice for $l(\omega)$ is a measure of how unlikely it is that ω holds in the environment. This ordering enables the planner to build reactor adaptations to deal with relaxation of the least likely assumptions first. Only then it will take care of the relaxation of increasingly likely assumptions as it has time. This is the ordering we employ in our implementation (Section 8).

There is a surface similarity between this strategy and the anytime planning work [8]. The key difference is that an anytime planner produces a plan whenever it is asked (i.e., it is based on time), whereas this algorithm produces plans in "decrements" of ω (irrespective of time). We can adopt this approach because we do have a good measure of progress, namely ω, and because we do have the reactor to handle time-critical responses.

The process of changing one ω-ideal reactor to another in this 'chain' will typically involve a number of discrete structural changes to the reactor. Consider two adjacent ω-ideal reactors in the reactor improvement chain show in (9) above; the chain of partially complete ω-ideal reactors between these two is as follows:

$$\mathtt{R}^{\omega^i_0} \xrightarrow{\alpha^1_i} \mathtt{R}^{\omega^i_1} \xrightarrow{\alpha^2_i} \mathtt{R}^{\omega^i_2} \xrightarrow{\alpha^3_i} \ldots \mathtt{R}^{\omega^i_j} \xrightarrow{\alpha^i_{j+1}} \ldots \mathtt{R}^{\omega^{i+1}}$$

where $\mathtt{R}^{\omega^i_j}$ is the partial reactor containing the *jth* update to the ω^i-ideal reactor. We demand that each such partial ω-ideal reactor abides by the behavioral constraint that it behave at least as good as the previous ideal reactor.

$$(\mathtt{R}^{\omega^i_j} \mid \mathtt{EM}) \xrightarrow{\omega^{i+1} \wedge f(e^i_j)} (\mathtt{R}^{\omega^i_j} \mid \mathtt{GEM}) \tag{10}$$

Where e^i_j describes how 'incomplete' the reactor is (what work remains to be done on the reactor to bring it to the ω^{i+1}-ideal reactor) and $f()$ maps this onto a set of evolution conditions. We place the following constraints on e^i_j

$$\begin{array}{l} \omega^i = \omega^{i+1} \wedge f(e^i_0) \\ e^i_j > e^i_{j+1} \\ \forall\ i\ \exists\ k > 0\ \ s.t.\ e^i_j = 0 \end{array} \tag{11}$$

These conditions allow us to guarantee that the planner can guarantee convergence from the ω^ith-ideal reactor to the ω^{i+1}th-ideal reactor.

The ω-ideal reactor description (8) only specifies what happens *if* ω holds. A safety constraint is also necessary to guarantee reasonable behavior in the case that some assumptions in ω don't hold. Let $\mathtt{EM}^\gamma$ be the process model of the environment in which the set of assumptions γ holds. By a *changing environment* we mean one in which γ is not constant. The ideal reactor constraint can now be written as

$$(\mathtt{R}^{\omega} \mid \mathtt{EM}^\gamma) \xrightarrow{\omega \subseteq \gamma} (\mathtt{R}^{\omega} \mid \mathtt{GEM}^\gamma) \tag{12}$$

If $\omega \not\subseteq \gamma$ then that part of the reactor that relies on the assumptions in $\omega - \gamma$ is compromised. For the planner to revise these parts the reactor must be capable of noticing that the assumptions in $\omega - \gamma$ have failed. Such a reactor will be referred to as an *observant* reactor, and from now on that we assume we are always working with an observant reactor. An additional constraint for convergence is thus that the planner be able to change the relaxation ordering to incorporate the immediate relaxation of a failed assumption, i.e., that the planner be capable of relaxing assumptions in *any* order. Notice that reintroduction of an assumption into the environment is not a problem – the ω relaxation algorithm is monotonic with respect to assumptions.

An important timing issue, related to the previous discussion of anytime planning, but not dealt with in this paper, is the issue of guaranteeing the planner response time. The spirit of this work has been to position the time-critical code in the reactor, and leave the planner with minimal time-constraints. For this reason, rather than produce a time-constrained planner, we have addressed the issue of making the reactor operate safely *while* changes are in progress.

While the planner is working on revising the reactor, it's necessary to ensure some reasonable behavior from "faulty" components. This is a difficult constraint to implement generally. What is really needed is the guarantee that the faulty reactor won't do anything to the environment that would prevent any future ω-ideal reactors from working by, e.g., breaking some crucial component. We settle for a more restricted constraint here: that the reactor exhibit no behavior based on the faulty assumptions. This can be implemented simply by disabling all reactions that depend on the invalidated assumptions $\omega - \gamma$.

7 The Planner

We begin this section with a description of the basic planner algorithm. More detail is then presented on some of the unique aspects of the algorithm, such as reducing planner 'expectations' and generating perceptions. The planner is constructed on an Interval Temporal Logic (ITL) reasoner, and this reasoner is briefly presented. The section concludes with the verification that the planner meets the convergence constraints presented in the previous section.

7.1 Planner Algorithm

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

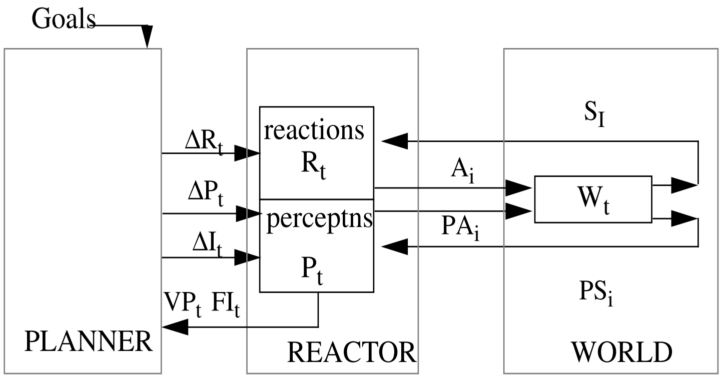

Figure 3: Planner-Reactor Interactions.

reactor to achieve G_t. At every iteration, the combination of the reactor model R_t and the expectation is exactly the ω-ideal reactor (where ω is the current set of assumptions the planner is working with).

This expectation is what we map to the e^i_j function in the previous section, equation (11). The process of incremental improvement of the reactor can now be seen as 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 ω-ideal reactor. Additionally, when assumptions are used to construct the reactor increment, the planner will insert assumption monitors ΔI_t into the reactor. Similarly, if the planner needs specific information for its own planning purposes, it can insert additional perception processes ΔP_t, processes that collect information in the reactor and report back to the planner.

The input/output interactions between planner and reactor are as shown in figure 3. 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 block diagram of the planner's internals is shown in figure 4 and explained here. Based on the environment model EM and the goals G_t, the planner accumulates a current Problem-Solving Context (PSC). 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 analyzed 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. A set of perception requests can also be generated at this point ΔP_t; these requests will return data that clarify uncertainty in EM_t and allow further decomposition of E_t. The assumption monitor processes may signal that the assumptions they protect have been violated; this results in $Assm_t$ being decremented by that assumption, and it may thus result in E_t being revised and probably increased again.

7.2 Expectation Reduction

To reduce an expectation, the planner reasons within a Problem Solving Context consisting of the relevant parts of the environment model EM, the action repertoire A and the assumptions $Assm_t$. The outcome will be a reactor adaptation ΔR_t, the reactions necessary to implement the expectation reduction ΔE_t.

The manageable size of ΔE_t is constrained by the following factors:

1. The response time of the planner. This can come from the design specification of the planner, or from the temporal constraints on goals and actions in the plan.

2. The effects of uncertain knowledge on the plan; the planner may need to issue perception requests before more of the expectation can be reduced.

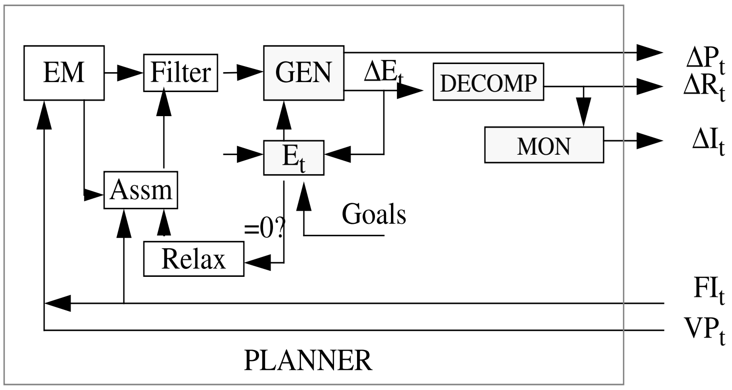

Figure 4: The Planner Architecture.

3. Resource use and mappings.

Initially, the planner reduces an expectation by looking for an abstract plan that achieves the given goals. This abstract plan, representing the structure of a more detailed plan, is maintained through insertion of situations into the reactor. The more detailed actions corresponding to the expansion of an abstract action are encoded as reactions in the situation, representing the abstract action. Any reactor segment sent to the reactor must be directly executable. However it need not necessarily be *concrete*, i.e., a reaction of a given situation may only contain a `STUB` process, a process that acts as a 'placeholder' and, when executed, only sends a perception back to the planner stating that this particular situation has become active. We call this signal a *stub-trigger*. In this way, the planner can adapt the reactor without needing to fully refine its plan first.

Before inserting any adaptation into the reactor however, the planner proceeds to detail the first step in the abstract plan, such that for this first step a concrete plan is in place (i.e., no `STUB`) when the adaptation is sent. This avoids an immediate stub-trigger signal from the reactor for this segment. Instead it allows the reactor to start operating immediately, while the planner can proceed in parallel to flesh out other abstract parts of the plan.

Barring any perceptions received from the reactor, the planner continues this process, incrementally fleshing out abstract segments. At every iteration, the expectation is reduced by fleshing out a situation. The situation hierarchy is restricted to be an acyclic digraph, i.e., abstract actions used higher in the plan hierarchy cannot be used to construct a concrete reaction.

If all the reactor segments are made concrete and all the `STUB` processes removed. Then, the planner has achieved an ω-ideal reactor, and can proceed to relax the next assumption.

Constructing adaptations. Once the planner has generated a possibly partially abstract plan, it is ready to send it to the reactor as an adaptation. The structure of the abstract plan is preserved through the definition of a situation hierarchy (in the flat representation). An adaptation contains the definition of new reactions (the 'P' processes), and the steps needed to safely change over a situation from the old to the new reactions.

If any of the new segments rely on assumptions, the adaptation will contain additionally an assumption monitor, a process that repeatedly checks whether the assumption still holds in the environment, and, if not, will signal a perception back to the planner.

7.3 Perceptions

Perceptions may change the course of operation of the planner. They provide key information for the planner to decide what part of the reactor to elaborate upon next. Apart from informational perceptions, the planner relies on four specific perceptions to set its priorities for the planning process. These perceptions are the following:

1. Guard-kills. When an `E` process (a 'guard') is killed, the planner is informed. `E` processes form the basis by which the planner can remove old structure and install new structure in the reactor. This perception indicates the completion of an adaptation to the planner.

2. Stub trigger. The execution of a `STUB` process in the reactor signals the planner that the reactor has started to execute a segment for which no concrete plan yet exists.

3. Situation failure. This signals the planner that a reactor segment failed (usually through an invalidated assumption), and needs to be modified as soon as possible.

4. Assumption failure. The planner installs dedicated processes (ΔI_t) in the reactor to monitor assumptions. Should one of the assumptions be invalidated, the associated monitor will signal the planner.

Guard-kills govern the adaptation cycle and are necessary for the planner to preserve the safety of the adaptation process. If a situation is still being adapted, the planner cannot send out a next adaptation for that same situation since the safe adaptation mechanism is only safe for one adaptation per situation at the time. Thus the planner holds back the next adaptation instructions for that situation until the guard-kill perception is received.

The other three types influence the planner's priority in planning adaptations. The planner uses these to decide which part of the expectation to reduce first. In effect, the environment (through the reactor) tells the planner what parts of the reactor should be considered.

The priory ordering for planning is the following:

$$Failed\ situations\ >\ Forced\ assumption\ relaxation > \ Active\ stubs > \ Quiescent\ stubs.$$

The planner relies on situation failure and assumption failure signals to "understand" what is happening in the environment. The general problem of deducing that an assumption has failed, or determining why a situation has failed, is very difficult and might require complicated error models or causal reasoning. For our implementation, we have restricted outselves to a set of assumptions whose validity can be reliably assessed with direct measurements. In similar spirit, we interpret situation failures as evidence of assumption failures. We justify making this simplification by noting that the issues of deducing assumption failures could be tackled independently of this work on incremental adaptation. It is also, however, an area we wish to study further.

<u>Forced Relaxation.</u> Any of the assumption monitors or stub triggers in the initial reactor segments can potentially signal the planner and divert its attention from the *a priori* established ordering of assumptions relaxation. Failure of a situation to successfully complete its reactions is also cause for a perception. These three sources of perceptual input have different effects on the planner.

On receiving a stub trigger signal, the planner redirects its focus of attention for refining the abstract plan to the portion of the plan containing the stub trigger.

An assumption failure perception has a larger effect than simply refocusing attention. The assumption failure signal causes the planner to negate the assumption in its PSC and begin to rebuild the effected portions of its plan. If a situation relying on that assumption failed, that particular situation is given highest priority for adaptation, even though other assumptions may have priority in the assumption ranking. This means the planner may have to turn its focus back to parts of the plan it had previously considered finished.

Apart from its cause, a forced relaxation or refinement results in the same reactor segment and adaptation sequence as a normal relaxation or refinement. Once the planner has achieved a complete ω-ideal reactor, it is ready again to pop up to the outer loop and select an assumption for normal relaxation.

```
atp_actionschema Lacks <type,a><>
intervals i1,i2,i3,i4,i5;
    applicability:  AVAILABLE(type)     @ i1 and ITL(i1, encompasses, Lacks);
                    ACT ()              @ i2 and ITL(i2, encompasses, Lacks);
    executability:  PART(a,tray)        @ i3 and ITL(i3, encompasses, Lacks);
                    POSITION(a,trayarea)@ i4 and ITL(i4, encompasses, Lacks);
    effects:        INTRAY(type,a)      @ i5 and ITL(Lacks, meets, i5);

    expansions:     Exists x,i1,i2:
                        PART(x,type)    @ i1 and ITL(i1, encompasses, i5)
                    and IN(x,a)         @ i2 and ITL(i2, encompasses, i5);
endactionschema;
```

Figure 5: Example (abstract) action schema.

7.4 The Kernel of the Planner

The kernel of the planner is an Interval Temporal Logic (ITL) reasoner, based on Pelavin's logic [23]. The planner architecture is built on top of this kernel. World models are maintained 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 [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., $ITL(i1, meets, i2)$ specifies that the interval $i1$ finishes just as interval of time as $i2$ starts. Other relations we use here include *equal* (both intervals denote the same time period), $disjunct$ (the intervals do not overlap in time), $finishes$ (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. Action schemas can be either abstract or basic. A basic schema represents a directly executable $\mathcal{RS}$ action schema, an abstract schema describes bigger chunks to quickly generate a plan outline, which later can be refined appropriate to environment specifics. The structure of the plan outline is retained in the reactor by encoding an abstract schema as a situation.

An example of an abstract action schema is given in fig. 5. This action schema `Lacks` has two input parameters *type* and *a*, and no results. Its specification uses five (extensionally qualified) intervals of time $i1$-$i5$, as well as the schema name, which denotes the interval of occurrence of the action in the specification. For search control reasons, the executability conditions are split into the executability conditions proper, conditions that the planner can backchain on, and applicability conditions, which have to be satisfied at action selection time. `Lacks` is an abstract schema, and therefore has *expansion goals*, besides its effects (a basic schema has no expansion or goals). In a plan refinement stage, these goals are additional goals that need to be satisfied.

A world model consists of a set of temporal predicates, describing the world in conjunction with a causal theory, describing consistency rules or derivable information. In fig. 6 two examples of causal theory rules are given. The first rule indicates that some types of parts are graspable, the second one specifies that an object can occupy only one position at the time. Finally, goals and assumptions are also specified as temporal predicates.

7.5 Convergence

This section now concludes with the verification that the planner does obey the convergence constraints described earlier. The block diagram in figure 4 is the basis of our argument. 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 assumption 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. This convergence constraint 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, we need to show that E_t can be reduced incrementally to zero. The initial expectation reduction is the construction of a first abstract plan from the goals. Subsequently, this abstract plan is incrementally refined with its structure maintained in the reactor as a situation hierarchy. The refinement proceeds a situation at the time, and since the hierarchy is limited to be an acyclic digraph, no recursion can occur. Given a finite set of abstract actions, this implies a finite depth for any situation hierarchy. If every individual refinement can be planned for, or shown to be impossible given the information available to the planner, then eventually the whole plan will be made concrete in finite time. Nonetheless, specification of impossible goals will result in a reactor that contains only a partially concrete situation hierarchy.

8 The Kitting Robot Implementation

Up to this point, the planner-reactor concept has been presented in a more or less domain independent fashion. This section describes the implementation of our planner-reactor based robot kitting workcell and provides some detailed examples of the system in operation.

8.1 Review of the Kitting Robot Problem

Assembly components are fed to the kitting robot on a conveyor belt. The robot must take parts off the belt, place them in the appropriate slots of a kitting tray, and then place the kitted tray onto the belt. The trays are stored in a stack in the robot's workspace. The product we are kitting is a small DC servomotor sold by Philips. It has three parts: a cap, a motor and a body, all of which have a number of variants (see figure 7).

The purpose of the kitting robot is to produce kits, filtering out the various sources of uncertainty in parts supply. The parts supply may be uneven and biased towards one type of part (starving others). The parts may be of mixed quality. It may be possible to substitute one part variant for another (if running mixed batch). Parts may move around while on the belt. The kitting tray may be disturbed while kitting is in progress: it may be moved and/or parts may be added or removed. Furthermore, automation downstream of the kitting robot may or may not be in a position to accept more trays and depending on factory policy in-line buffering may or may not be allowed. The kitting robot needs to counter each of these sources of uncertainty. Furthermore, the kitting robot has to respond in a timely fashion to task changes issued by factory management.

8.2 Kitting Assumptions

The planner-reactor approach provides a way to incrementally construct reactive systems with improving performance. At the heart of this iterative mechanism is the concept of characterizing the environment by a set of assumptions and methods to detect whether they hold or not. In the case that the environment goes through regimes where different subsets of assumptions hold, the planner can capitalize on this, especially when new factory goals have been received and resultant changes in behavior

```
;;; some parts are graspable
Forall x,y,i: PART(x,y)  @i and ( y = body  or y = cap or
                                  y = motor or y = tray )
        => GRASPABLE (x) @ i ;

;;; only one position can be held at once
Forall x,y,z,i1,i2:   POSITION(x,y)@i1 and POSITION(x,z)@i2 and y /= z
        => ITL(i1, disjunct, i2)
```

Figure 6: Example consistency rules.

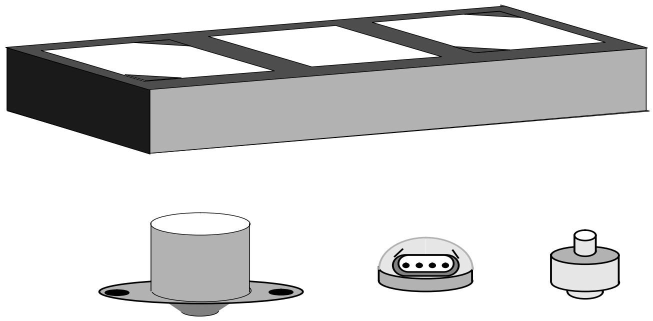

Figure 7: The Kit Tray and Parts.

are necessary. The planner uses these assumptions to help it to quickly build a reasonable reactor. Later, as it has time, it relaxes assumptions and fashions improved versions of the reactor.

The set of assumptions we have developed for the kitting environment is described below in reverse order of likelihood of holding; i.e., in the order in which the planner considers relaxing them. Some of the assumptions, such as that of the quality and substitutability of parts, *must* eventually be relaxed to get to a robust workcell. Some other assumptions, such as that of cooperation in kit assembly and tray disturbance, describe contingencies that make a more versatile workcell, but are not necessary for a robust system. Still other assumptions pertain to unusual operating conditions, such as the assumptions of no competition and no downstream disturbances.

1. Assumption of Parts Quality (AQ): all the parts coming into the kitting workcell are of good quality and do not need to be tested. When this assumption is relaxed, the robot must test parts before they are used, and reject bad quality parts.

2. Assumption of non-substitutability of parts (AS): each part has only one variant. When it holds, it rules out having to reason about product variants and mixed-batch kitting.

3. Assumption of unfilled trays (AF): all trays arrive empty. This means the robot doesn't have to check trays when they arrive. If trays are being routed back to this cell as rejects from downstream manufacturing, then the robot has to determine which new parts each tray needs.

4. Assumption of no competition (ACT): parts placed into a tray stay there. This will only be violated in the case of contentions between multiple workcells or faults in the transport system.

5. Assumption of no cooperation (ACP): the robot is not sharing the job of populating kits with other workcells. Thus, it only expects kits to be finished when it explicitly finishes them.

6. Assumption of no tray disturbance (ADT): trays remain where they were put. Again, this would only be violated if the kitting job is being done cooperatively between several robots or between humans workers and robots.

7. Assumption of no parts motion (AM): the kit parts do not move around once delivered on the belt to the workcell. This means the robot can take one snapshot of the scene at the start of the kitting sequence and the part positions will be valid throughout.

8. Assumption of no downstream disturbance (ADD): downstream automation is always ready to receive finished kits. If downstream automation signals a problem then ADD is withdrawn, and it is reinstated when downstream automation signals that it is again willing to accept kits.

9. Assumption of parts ordering (AO): the kit parts must be placed in the kit tray in a particular ordering. This gives the kitting robot the ability to modify parts ordering should that be warranted.

8.3 The Kitting Task Implementation

In this section, we will work through an example of adapting a kitting reactor based on goals. We will initially assume what we refer to as a *null reactor* – a reactor with only the basic sensory and exploratory behaviors necessary to support the kitting task. The first step in our adaptation example will be the reception of new goals — the goal to begin constructing the servomotor kits — from factory management.

New Goals. The goal received from factory management is:

$$\begin{aligned} &\forall t, i1 : Instance(tray, t)@i1 \Rightarrow \\ &\qquad \exists i2 : (FullTray(t) \wedge ToDownStream(t))@i2 \wedge ITL(i2, finishes, time) \end{aligned} \tag{13}$$

This conditional goal specifies that whenever there is an instance of a tray, then 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 $FullTray(t)$ can be written out as:

$$\begin{aligned} FullTray(t)@i1 \iff \quad & [\ \exists x, y, z Instance(cap, x) \wedge Instance(motor, y) \\ & \wedge Instance(body, z) \wedge Contains(t, x, y, z)\ ]@i1 \end{aligned} \tag{14}$$

The planner begins by constructings its Problem Solving Context (PSC), as described in section 7. This is the 'frame of mind' in which it is going to 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 `Find` (to locate an instance of an object model using vision) and `Get` (to acquire an object); $Contains$ brings in a number of subgoals, such as $In(t, x)$. In addition, the domain rules may specify default assumptions. In this example, the action `Get` brings in the assumption of no object motion AM. `Find` brings in both the assumptions of good parts quality AQ and no substitutability between parts, AS. Although the planner does not know whether any of these assumptions actually do hold in the current environment, it always begins by assuming they do and asserting them in its PSC.

The planner invokes its ITL reasoner to attempt to construct a series of actions from its PSC that will achieve the goal. Chaining from effects to preconditions may drag in new actions and new assumptions. The planner will proceed with working on its abstract plan in this fashion until it has found an action that can be executed immediately. In that case, the planner determines how much of the abstract plan is already in the reactor and how much needs to be added via adaptations. This 'difference,' the new reactor structure that needs to be added in to implement the current abstract plan, we will refer to as a *reactor segment.*

All portions of a reactor segment must be executable. Any parts of the abstract plan that are not immediately executable are replaced in the reactor segment with *stub trigger* processes. These processes will warn the planner should the reactor need these reactions fleshed out. Any part of the plan that relies on assumptions is equipped with *assumption monitor* processes. Should the monitored assumption not hold when the relevant reactions are active, then the planner is informed.

Generation of the First Adaptation. Regressing on the conditional goal gives us the first abstract description of the reactor segment: first locate any tray, then fill that tray, and finally move the tray to the belt for transfer to downstream automation. The first immediately executable action is the action to acquire a kitting tray.

Recall that the reactor is represented as a set of concurrent situation-triggered reactions as shown in equation (2). The planner defines the reactor by constructing a set of situations and for each situation a set of reactions that should be enabled in that situation. Situations can be hierarchical in that reactions for one situation may assert a second.

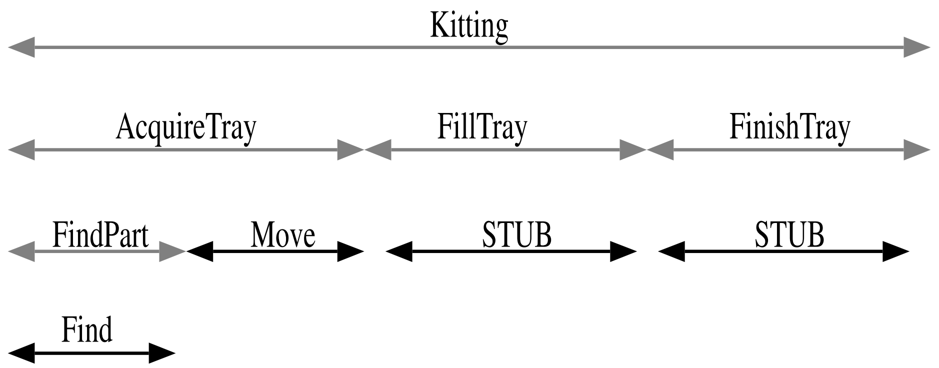

Figure 8: Initial Adaptation to Kitting Reactor.

The process descriptions below show a reactor segment generated by the planner for this initial situation[1].

$$
\begin{array}{lll}
\mathtt{P0} & = & \mathtt{STOP} :; \mathtt{ASSERT}_{Kitting} \\
\mathtt{P1} & = & \mathtt{SIT}_{Kitting}\langle k\rangle :; \\
 & & \quad (\ \sim((\mathtt{REASSERT}_{AcquireTray}\langle x1\rangle : \mathtt{REASSERT}_{FillTray,x1} : \mathtt{REASSERT}_{FinishTray,x1}) \\
 & & \qquad : \mathtt{SUCCEED}_{Kitting,k}) : \mathtt{FAIL}_{Kitting,k}\ ) \\
\mathtt{P2} & = & \mathtt{SIT}_{AcquireTray}\langle k\rangle :; \\
 & & \quad (\ \sim(\mathtt{REASSERT}_{FindPart}\langle x1\rangle : \mathtt{Move}_{x1,trayarea} \\
 & & \qquad : (\mathtt{UPDATE}_{AcquireTray,k,x1}; \mathtt{SUCCEED}_{AcquireTray,k})) : \mathtt{FAIL}_{AcquireTray,k}\ ) \\
\mathtt{P3} & = & \mathtt{SIT}_{FindPart}\langle k\rangle :; \\
 & & \quad (\ \sim(\mathtt{Find}_{tray}\langle x1\rangle \\
 & & \qquad : (\mathtt{UPDATE}_{FindPart,k,x1}; \mathtt{SUCCEED}_{FindPart,k})) : \mathtt{FAIL}_{FindPart,k}\ ) \\
\mathtt{P4} & = & \mathtt{SIT}_{FillTray}\langle k, x1\rangle :; (\mathtt{STUB}_{FillTray} : \mathtt{SUSPEND}_{2000}; \mathtt{FAIL}_{FillTray,k}) \\
\mathtt{P5} & = & \mathtt{SIT}_{FinishTray}\langle k, x1\rangle :; (\mathtt{STUB}_{FinishTray} : \mathtt{SUSPEND}_{2000}; \mathtt{FAIL}_{FinishTray,k})
\end{array}
$$

Figure 8 shows the situation hierarchy for this reactor segment. In the figure, situations are shown as lightly colored arrows. The lowest levels of the situation hierarchy contain only reactions. These are shown as darker arrows. The situation hierarchy figures are read as follows. The topmost line is the highest level situation (e.g., 'Kitting' for this initial reactor). The line underneath this indicates how this situation is broken down hierarchically. For example, in this initial reactor, Kitting is broken down into a sequential composition of first 'AcquireTray' then 'FillTray' and then 'FinishTray.' Recall that activating a situation only means that a particular set of reactions now become enabled. In 'AcquireTray,' these are the reactions to bring a tray into the workspace; in 'FillTray' they are the reactions to place all the parts into the tray. The actual behavior of the robot in a situation will simply depend on what choices the environment offers. In this initial reactor, the 'AcquireTray' situation is in turn composed of a 'FindPart' situation followed sequentially by a `Move` reaction. The situations 'FillTray' and 'RemoveTray' simply have the placeholding `STUB` reactions in them.

Each reaction in the segment is in the special conditional form ($\sim(\mathtt{Cond} : \mathtt{IfTrue}) : \mathtt{Else}$) introduced in section 4 and employs the `REASSERT` process to continually reassert failing situations. Note that `P0` is necessary to assert the topmost situation repeatedly. The reactor segment uses three reactions: The process $\mathtt{Move}_{obj,loc}$ repositions a grasped object *obj* to a location *loc*. The process $\mathtt{Find}_m\langle p\rangle$ locates and acquires an object of type m and returns a pointer to the sensory model for this instance of the object type. The process `STUB` simply sends the stub trigger signal back to the planner.

[1]All the reactor segments in this paper have been simplified to omit statistics collection and other inessential processes that are present in the actual implementation.

Assumption Monitors. All the assumptions used by the planner in coming up with this first section of code have to be monitored. In our current implementation *all* actual assumption monitoring, *except* for the motion assumption, has been bypassed. To monitor an assumption, the planner creates an instance of the `ASSUMONITOR` process giving as arguments the name of the assumption to monitor (using the acronyms introduced earlier) and how often (in milliseconds) to check it. Checking an assumption reduces (except for the motion assumption) in our implementation to simply checking a flag. An external agent can signal that an assumption has been violated by setting the appropriate flag. If there is a monitor process for that assumption it will eventually discover the assumption failure and send a signal back to the planner to let it know the assumption has failed. In contrast to this approach, the assumption monitor for the motion assumption AM will report assumption failure when the robot fails to find an expected part at its expected positions.

The planner produces the following assumption monitors for the first adaptation:

$$
\begin{aligned}
\mathtt{P6} &= \mathtt{ASSUMONITOR}_{AS,Tcheck} \\
\mathtt{P7} &= \mathtt{ASSUMONITOR}_{AQ,Tcheck} \\
\mathtt{P8} &= \mathtt{ASSUMONITOR}_{AF,Tcheck} \\
\mathtt{P9} &= \mathtt{ASSUMONITOR}_{ACP,Tcheck} \\
\mathtt{P10} &= \mathtt{ASSUMONITOR}_{ADT,Tcheck}
\end{aligned}
$$

where $Tcheck = 2000ms$. This monitors will be added into the reactor in exactly the same fashion as the reactor segments `P0 - P5` using the safe adaptation procedure introduced in section 5.

Refinement to the first ideal reactor. Having issued these adaptations, the planner proceeds to deliberate further on the abstract plan. Each time it reaches an action that can be directly executed, it adapts the reactor to include the new portion of the abstract plan. For example, having completed the reactor segment above, the next concrete action that the planner reaches is in the refinement of the $FillTray$ situation, acquiring and placing the first component into the tray. The reactor segment resulting from this refinement is as follows:

$$
\begin{aligned}
\mathtt{P11} &= \mathtt{SIT}_{FillTray}\langle k, x1\rangle :; \\
&\quad (\ \ \tilde{}\,((\mathtt{REASSERT}_{LacksMotor,x1} \mid \mathtt{REASSERT}_{LacksCap,x1} \mid \mathtt{REASSERT}_{LacksBody,x1}) \\
&\qquad : \mathtt{SUCCEED}_{FillTray,k}) : \mathtt{FAIL}_{FillTray,k}\ ) \\
\mathtt{P12} &= \mathtt{SIT}_{LacksMotor}\langle k, x1\rangle :; \\
&\quad (\ \ \tilde{}\,(\mathtt{FindGetInsert}_{motor,x1} \\
&\qquad : \mathtt{SUCCEED}_{LacksMotor,k}) : \mathtt{FAIL}_{LacksMotor,k}\ ) \\
\mathtt{P13} &= \mathtt{SIT}_{LacksCap}\langle k, x1\rangle :; (\mathtt{STUB}_{LacksCap} : \mathtt{SUSPEND}_{2000}; \mathtt{FAIL}_{LacksCap,k}) \\
\mathtt{P14} &= \mathtt{SIT}_{LacksBody}\langle k, x1\rangle :; (\mathtt{STUB}_{LacksBody} : \mathtt{SUSPEND}_{2000}; \mathtt{FAIL}_{LacksBody,k})
\end{aligned}
$$

Here, the reaction $\mathtt{FindGetInsert}_{m,t}$ is a single, simple hardwired instruction to look for a part of type m and place it into the tray t. It ignores issues of part quality and so on. This reactor segment requires the replacement of the $FillTray$ situation previously loaded.

If the planner is not interrupted from this course by assumption failures or stub triggers, then eventually the entire abstract plan is fleshed out and sent via adaptations to the reactor. In our example, there are 3 more reactor segments produced after the two above, to fully flesh out a working kitting reactor: The two remaining $LacksPart$ situations and the $FinishTray$ situation.

The first ideal reactor, that is the first reactor that contains a full kitting task, is shown in figure 9.

Normal Assumption Relaxation. Once the planner has an ideal reactor established for a given set of assumptions (all assumptions initially), it then begins deliberation on relaxing the assumptions. The order in which assumptions are relaxed, apart from forced relaxation due to assumption failures, is the

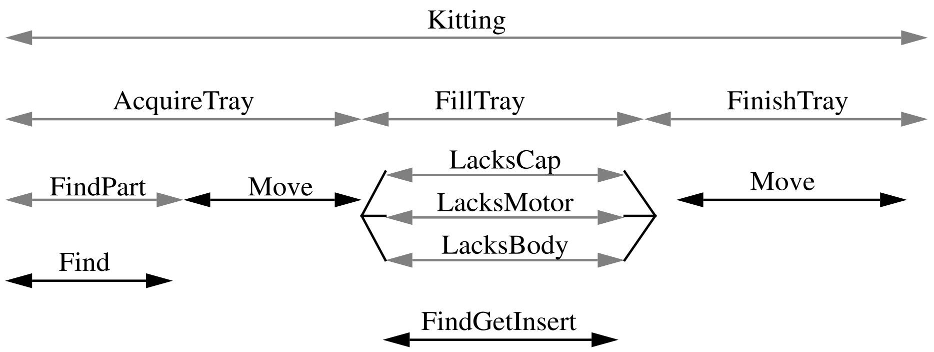

Figure 9: First Ideal Kitting Reactor.

order of likelihood of assumptions *not* holding, and is part of the input information to the planner. This ordering is the order that the assumptions are listed in section 8.2. Remember, of course, that while this is happening, the reactor is completing kits using the version of the kitting task constructed to date.

The first relaxation the planner considers is the assumption of quality (AQ). The planner relaxes the assumption by negating it within its PSC. This means that additional actions may now have to be planned to accomplish what previously had been assumed. In our example, the $FindPart$ situation is ruled out, and the planner constructs an alternate $FindGoodPart$ situation, which tests the acquired component to ensure it is of good quality; bad quality parts are rejected. The first reactor segment generated by the planner in considering this relaxation is the following:

$$
\begin{aligned}
\mathtt{P23} \;=\; & \mathtt{SIT}_{AcquireTray}\langle k\rangle :; \\
& \quad (\;\; {}^{\sim}(\mathtt{REASSERT}_{FindGoodPart}\langle x1\rangle : \mathtt{Move}_{x1,trayarea} \\
& \qquad : (\mathtt{UPDATE}_{AcquireTray,k,x1}; \mathtt{SUCCEED}_{AcquireTray,k})) : \mathtt{FAIL}_{AcquireTray,k}\;) \\
\mathtt{P24} \;=\; & \mathtt{SIT}_{FindGoodPart}\langle k\rangle :; \\
& \quad (\;\; {}^{\sim}(\mathtt{Find}_{tray}\langle x1\rangle : \mathtt{Test}_{tray,x1} \\
& \qquad : (\mathtt{UPDATE}_{FindGoodPart,k,x1}; \mathtt{SUCCEED}_{FindGoodPart,k})) : \mathtt{FAIL}_{FindGoodPart,k})
\end{aligned}
$$

This segment is a replacement for the $AcquireTray$ situation. Having made the incremental adaptations for this segment the planner continues to relax the assumptions for the $LacksMotor$, $LacksCap$, and $LacksBody$ situations also.

Relaxing all the assumptions produces a complicated reactor structure. This is partially shown in figure 10. This figure shows the full situation hierarchy when all assumptions are relaxed. The additional situations are tagged with the assumptions that cause them to be introduced. The comparison of this figure with the previous one, figure 9 emphasizes the advantage of incremental reactor construction: the reactor of figure 9 is faster to derive and implement then that of figure 10.

9 Results

9.1 The Implementation Environment

A Puma-560 robot is the basis of the kitting workcell. The original VAL-based controller is used to control the manipulator. The Puma is equipped with a four-fingered, 2-DOF pneumatic gripper. The VAL controller is connected by a serial line to one of a set of M68020 processors on a common VMEBUS. This processor functions as a robot server processor.

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

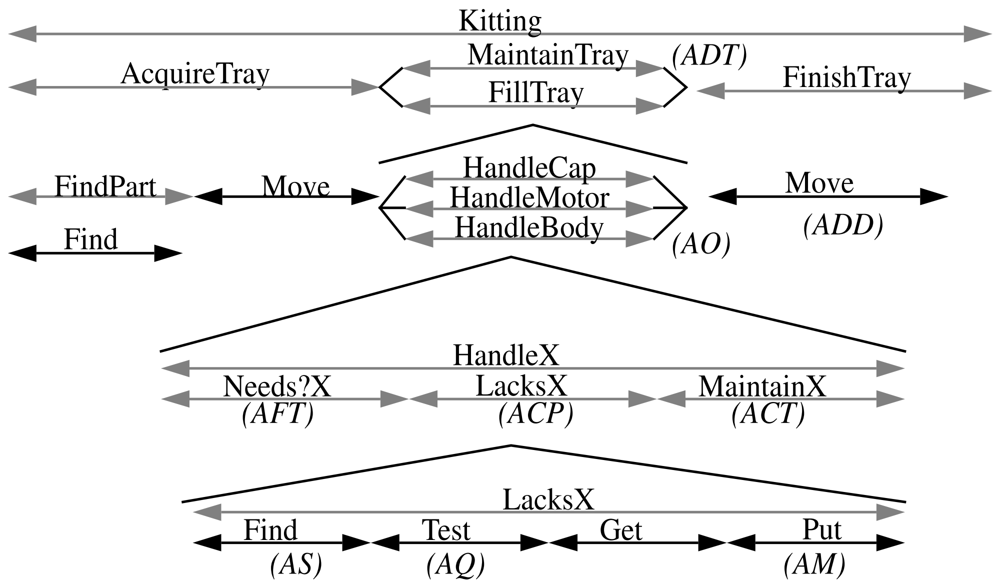

Figure 10: Final Ideal Kitting Reactor.

be used to get a close-up view of an object. 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.

The remainder of the cell consists of the worksurface and a conveyer belt on which parts can be introduced or removed from the worksurface. In the current implementation, the object arrival poses are restricted to lie within a specified subset of all possible poses. (This restriction is temporary.)

$\mathcal{RS}$ 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.

A set of basic sensory and motor schemas has been built to interface to the robot and vision servers. These 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 and only terminates when such a model instance is seen.

Planner-Reactor. The reactor is a set of $\mathcal{RS}$ processes running on the `RS-L3` system. Communication with the planner is via an ethernet UNIX socket link. The planner can load new process definitions

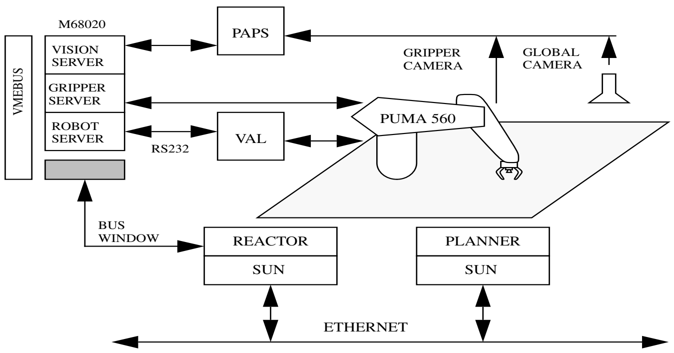

Figure 11: The Kitting Robot Implementation.

Figure 12: Graphs of adaptations issued and applied.

into the system over this link. Certain basic schemas such as `STUB, E, A` and so on, use this link to communicate with the remote planner.

The planner is implemented in the POPLOG environment. It employs the planning control strategy outlined in this paper and maintains the link with the reactor using asynchronous signal handling to catch incoming perceptions.

9.2 Experimental Results

The example described in the previous section was implemented and several experimental runs conducted. Trace statistics were gathered on each run on, e.g., the times when assumption failures were detected, when adaptations were made, the number of assumptions in use in the reactor, and so on. In each run, the planner utilized all the kitting assumptions. However, only five of the assumptions were actually relaxed in these trials. The purpose of these experimental runs was to begin to explore the behavior of a planner-reactor system in practice. Results from the runs are presented in the following subsections.

9.2.1 Construction of the Initial Reactor

Adaptations issued and applied. Figure 12 shows the trace of adaptations issued by the planner for the initial stages of the experimental run: the incremental construction of the first complete reactor. It takes the planner 26 seconds to issue the first adaptations. Subsequently, 16s after the first adaptation, the planner has completed all adaptations for that first complete reactor. The *adaptations issued* trace (the solid line) indicates the times at which adaptations are issued by the planner to `RS-L3`. An adaptation issued does not necessarily take effect immediately; we discuss this later. Each *adaptation issued* is a reactor segment such as those in the examples in section 8.3. Typically, a number of such adaptations are necessary to relax an assumption. The trace has a steep slope in periods of reactor change, and is flat otherwise.

As the safe adaptation procedure incrementally updates the reactor to include each new segment, it makes a number of discrete changes to the reactor. Each such change is logged as an *adaptation applied* (the dashed line in figure 12). The constraints involved in safe adaptation can cause a time lag in implementing changes. Active situations cannot be interrupted — that would leave the reactor in an undefined state — instead the adaptation waits for the situation to end before applying the changes, in accordance with the theory described in section 5.5. In general, the interval between an issued adaptation and an applied adaptation is quite small. If the situation to be changed is active however, the delay may be longer. Thus, the *adaptations applied* trace will lag the *adaptations issued* trace by a variable amount, e.g., the 2s lag between the first *adaptation issued* and first *adaptation applied.* There will always be at least one change in reactor structure (*adaptation applied*) per reactor segment (*adaptation issued*), hence the former quickly rises above the latter in the graph.

Assumptions in Use. Figure 13 is a trace of the number of assumptions in use by the planner and the reactor for this first phase of the experiment. The planner has initially no assumptions in use. As its goals drag subgoals and actions into the planner's PSC, the planner gathers the assumptions associated with those subgoals and actions. The policy of the planner is to assume that all assumptions hold, unless it actually knows otherwise (from perception data). The first reactor is in place roughly 28s after startup, and requires 6 assumptions (the start of the solid line in figure 13). Even though this reactor is not complete (in the sense that it contains `STUB` processes), the kitting robot can now start kitting; its kitting actions are overlapped in time with the further refinement of its kitting 'program.' In the process of initial construction of the reactor, the number of assumptions used by the planner increases gradually. The initial reactor is elaborated over the following 12s until by 40s after startup the first complete reactor is in place (i.e., the first ω-ideal reactor). This reactor employs all nine assumptions.

Figure 13: Graphs of Assumptions in use by planner and reactor.

Figure 14: Graphs of Assumptions in use by planner and reactor.

The dotted line in figure 13 is the trace of assumptions in use by the reactor. Assumptions introduced by the planner can only come into effect once the safe adaptation procedure allows it. Only then are the new segments in place that rely on that assumption. For the same reason that there is a time lag between adaptations issued and adaptations applied, there is a time lag between assumptions used in the planner and assumptions used in the reactor. In this initial phase of the experiment, the time lags are relatively small, 1–2s.

The planner finished the first working ideal reactor approximately 12 seconds after the first adaptation. This reactor is capable of repeatedly completing kitting trays as long as the entire set of kitting assumptions hold. In that 12 seconds, no `STUB` processes were triggered, indicating that the planner successfully refined the reactor *before* any unrefined sections were called on to execute. (This first reactor is the one shown in figure 8 in the previous section.)

9.2.2 Planner-Directed Relaxation of the Reactor

The next set of graphs show the planner relaxing the assumptions used to construct the first complete reactor. The order of relaxation in this experiment is the prespecified assumption relaxation priority. Consideration of forced assumption relaxation — when the environment forces the planner to relax an assumption prematurely — is relegated to the next section. When the planner decides to relax an assumption in its PSC, certain portions of its plan become invalid and need to be reworked. Figure 14 shows the trace of assumptions in use by the planner (solid trace) and reactor (dotted trace) for this second phase of the experimental run.

As soon as the planner finishes an ideal reactor for a given set of assumptions, it chooses the assumption that is least likely to hold (of the assumptions in use) for relaxation. For example, in figure 14, as soon as the planner finishes the first ideal reactor (at the leftmost edge of the short 'plateau' in the solid trace) it deliberates on relaxing the assumption of parts quality (AQ). The rightmost edge of the 'plateau' is the time that the planner finished issuing the adaptations necessitated by this relaxation. Thus the length of the plateau is the time taken for the planner to revise and regenerate the affected segments of the reactor plan (about 6s in this example). Similarly, the next highest 'step' to the right of the plateau describes the relaxation of the next assumption, the assumption of substitutability (AS), which takes about 30s.

The dotted trace in figure 14 is the trace of assumptions used in the reactor. This graph indicates how long assumptions remained in use. Once the first ideal reactor is constructed, each successive step in the trace indicates the duration in which successive ideal reactors were active. In the first phase of this run, the trace of reactor assumptions followed the trace of planner assumptions closely. However, in this second phase, although the planner has relaxed two assumptions in the period from 40s to 80s, the reactor does not begin to echo these changes until 103s. The reactor then finishes *both* relaxations between 103s and 104s. Thus, it has taken the reactor roughly 60s to complete the necessary changes. These lags and overlaps in operation underscore the highly asynchronous and independent operation of planner and reactor.

Figure 15: Graphs of Planner Assumptions and Adaptations Issued and Applied.

Figure 16: Graphs of Reactor Assumptions and Adaptations Issued and Applied.

Figure 17: Graphs of Assumptions in use by planner and reactor.

The cause of this delay is again the safe adaptation procedure. The situation is illustrated by the graphs in figure 15 and 16: a trace of adaptations issued and applied is superimposed, first on the trace of planner assumptions, and then on reactor assumptions. The planner's activity can been seen as clearly defined steps in the trace of adaptations issued. However, the safe adaptation mechanism causes these changes to become spread out in the trace of adaptations applied. The interval between 40s and 103s on the trace of adaptations applied in fig. 15 indicates there are changes in progress in the reactor throughout this period.

It is important to note again that the kitting robot is active and producing kits during this process; indeed it has been active since the first adaptation. The result of assumption relaxation is iterative improvement of the range of environmental conditions the kitting robot can handle; events that may cause an early ideal reactor to fail to assemble and thus reject a kit, will cause no problem for later reactors.

9.2.3 Environment-Directed Relaxation of the Reactor

In constructing a reactor, the planner is not aware of the exact state of affairs in the environment. This is a consequence of the division of responsibilities between planner and reactor. The reactor is the portion of the system that directly measures and responds to the environment; the planner is the portion of the system responsible for global reasoning. Nonetheless, the planner does need to have some information from the environment. It obtains these perceptions through the reactor.

Especially, the planner needs to know when an assumption that is in use in the reactor does not hold in the environment: assumption failure signals. Another perception that the planner needs to know is the case when the reactor is being called upon to execute parts of its code that have not been completely refined by the planner: stub trigger signals. As we have previously mentioned, although the planner does incorporate all the kitting assumptions when building a reactor, only the AQ (quality), AS (substitutability), AM (motion) and ADD (downstream-blocked) were used in these experimental trials. When an assumption failure occurs, that failure is repeatedly sent to the planner until the planner disables that portion of the reactor via an adaptation.

To determine whether an assumption holds, the planner needs to provide a condition that the reactor can measure and report back to the planner. For assumptions such as AQ and ADD, this condition is simply a monitor for an incoming signal. Downstream automation can be reasonably expected to report a problem with either of these assumptions to the kitting cell. The condition to invoke substitutability of parts, AS, is that the cell be starved of a particular object type. However, in our implementation sofar, failure of this assumption is simply a message sent to the workcell. The Motion assumption, AM, is the only one that the kitting robot actively measures. This assumption is considered to have failed if the kitting robot fails to acquire an object at the location where vision revealed an object. In the following graphs, AQ and AS have already been relaxed by the planner, and AM and ADD will be relaxed due to assumption failures signals.

Figure 17 is a graph of the assumptions in use in reactor and planner for the final phase of this experimental run. It additionally shows the trace of *assumption failure* signals received. Intervals of assumption failure are indicated by a rising slope. The flat regions between these slopes indicate intervals in which one or more ideal reactors operate successively without assumption problems. Just after 150s, the assumption monitor for the no-motion assumption (AM) begins to signal an assumption failure (triggered by removing a kitting part just as the robot was ready to close its gripper). The planner responds by relaxing the AM assumption (solid line) and the reactor quickly follows suit (the dashed line). A second failure and forced relaxation happens at 215s for the downstream automation assumption (ADD; triggered by a direct signal to the reactor).

Figure 18: Graphs of Adaptations, Assumptions and Assumption Fails.

Figure 18 shows a close up of the traces for adaptations issued and applied, failures, and the assumptions in use in the reactor, for this third phase of the experimental run. At 152s the AM assumption failure is signaled. The planner responds very quickly and begins to issue adaptations by 155s, continuing for about 10s. There is little delay in the safe adaptation procedure and the structural changes to the reactor lag the adaptations issued by only about 1s. This is also evident from the assumptions trace in 17. Part of the changes in this adaptation is to remove the assumption failure monitor for this assumption; thus, partway into the adaptation (161s) the signals from the assumption monitor cease. The total forced relaxation response time — the time from failure to final change in the reactor — is roughly 11s. The ADD failure occurs at 215s and proceeds in a similar fashion.

Reinstating Assumptions. In our theoretical analysis in this paper, we did *not* address the problem of reasserting previously failed assumptions. Nonetheless, this is convenient in practice to model operating regimes, and therefore is part of our objectives. In the experimental run, at time 357s the ADD assumption was reinstated. On the relaxation of such reinstatable assumptions, the planner adds a reactor monitor that will signal when the assumption holds again. (It doesn't show up as an assumption failure, since it isn't.) As the trace statistics show, our implementation does indeed handle the reintroduction of assumptions. However, further theoretical work is necessary to extend our convergence results to include this case.

The examples in the previous graphs purposely isolate forced relaxation from normal relaxation, and isolate relaxation of individual assumptions. It is, of course, possible for forced and normal relaxation to be mixed and for multiple assumptions to be relaxed simultaneously. This kind of mixed behavior is more what we would expect from an actual factory environment.

Performance Improvement. As the planner continues to refine the reactor, the overall kitting robot system should show improving efficiency and robustness. Ideally, to evaluate an implementation, an empirical measurement that quantifies performance should be collected for the duration of an experimental run. Unfortunately, in a dynamic and uncertain environment, simply because an action *may* fail, does not mean it *will* fail in any given trial. Thus, such a performance measurement must be made over a statistically significant number of trials. Practical difficulties prohibit us from collecting this number of trials on our PUMA testbed.

An indirect performance improvement measure can be formulated based on assumption use, and used to illustrate the ongoing improvement. Figure 19 shows a graph of such a measurement for the same experimental run described in the previous graphs. The performance measurement is graphed as $\frac{1}{M}$ versus time, where M has three components:

- The number of assumptions used in the planner (#SAP) and reactor (#SAR) that *also* hold in the environment.
- The number of assumptions retracted by the planner (#RAP) and in the reactor (#RAR).
- The change in number of assumptions changed in the environment from time $t-1$ to time t ($\#AE_{t-1} - \#AE_t$).

Since retractions affect the first component of M *as well as* the second, its necessary to multiply the second component by 2 to 'cancel' out the effect on the first. The formula for M at time t is:

$$M = (\#SAP + \#SAR) + 2(\#RAP + \#RAR) + (\#AE_{t-1} - \#AE_t)$$

Overall M is bigger for systems that have more supported assumptions and more assumptions retracted – an improved system. The reciprocal of M is taken to put this measurement in the typical form of a 'learning curve'.

The overall trend in figure 19 is clear: the performance improves over time. The initial decrease comes from the initial construction and then planner-directed assumption relaxation of the reactor. The

Figure 19: Planner-Reactor Improvement Graph.

'setback' in improvement at times at 152s and 215s are due to assumption failures. The 'setback' at time 357s is due to the reintroduction of the ADD assumption.

10 Conclusions

This paper has presented a general solution to integrating reaction and deliberation. A central concept of this integration is the asynchronous interaction between planner and the separate and concurrently operating reactor to maintain reactivity at all times. An incremental reactor construction and assumption relaxation strategy was also presented. The reactor was designed using a formal model of computation, $\mathcal{RS}$. This allowed us to develop a safe adaptation mechanism, to perform a formal analysis of the reactor adaptation process, and to prove convergence of the iterative effects of planner adaptations.

We have illustrated this solution with an industrial problem, the kitting robot. Based on our prototype implementation, we have presented quantitative performance results – a first for integrated systems of this kind. The results show the feasibility of this approach: an initial working reactor is quickly generated and correctly generalized over time. External events, such as assumption failures, are handled swiftly with non-affected reactor segments not being interrupted during the changes required.

Our system, as it stands, has a number of flaws.

1. We assumed a simple approach to monitoring assumption failures: All failures must be directly observable. In future work, we plan to tackle the issue of observability in relation to assumption failures. Error recovery capabilities will be needed to handle non-observable assumptions.

2. Our formulation of reactor behavior in the case of a reactor in which some assumptions no longer hold (i.e., while the planner is revising the reactor) has a loophole. We disable any reactions that are based on invalid assumptions. Doing nothing is not, however, always the right thing to do. Consider a mobile robot that detects a chasm ahead, but its avoidance reactions have been disabled because they rely on a faulty assumption. In such cases, it may be necessary not only to disable faulty reactions, but also enable "default" safety reactions.

3. The ω relaxation strategy is monotonic, and does not take advantage of assumptions that are re-introduced into the environment. We need to extend our theoretical treatment to model 'operating regimes', where assumptions can be reinstated over time. This will affect the convergence results for assumption relaxation.

Although our solution was driven by the kitting robot problem, the approach is generally applicable. It is potentially applicable in any domain in which both reaction and deliberation are required: exploration in mobile robots, emergency response planning, or advanced 'teleoperation' in planetary rovers. More specifically, for our solution to be applicable to a problem domain, the following should hold: First, it must be possible to identify a set of assumptions that characterize the environment. Second, it must be reasonable to assume that at any time, a nonempty subset of these assumptions will form a temporary stable operating regime. Problem domains in which these two don't hold effectively require the ideal reactor to be in place from startup, and thus present no advantage to our incremental approach.

References

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

[2] M.A. Arbib. Schema theory. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence (2nd Edition)*. Wiley-Interscience, 1992. Also USC Center for Neural Engineering TR-91-03.

[3] R. Arkin. Integrating behavioral, perceptual and world knowledge in reactive navigation. *Robotics & Autonomous Systems*, 6(1,2):105–122, 1990.

[4] 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.

[5] E. Brinksma (Project Editor). *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior.* OSI Project 97.21.20.2, 1987.

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

[7] J. Connell. Sss: A hybred architecture applied to robot navigation. In *IEEE Int. Conf. Robotics & Automation*, pages 2719–2724, Nice, France, May 1992.

[8] T. Dean and M. Boddy. An analysis of time-dependent planning. In *IJCAI-87*, pages 49–54, 1987.

[9] 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.

[10] R.J. Firby. Adaptive execution in complex dynamic worlds. Ph.D. Dissertation & Research Report YALEU/CSD/RR#672, Yale University, New Haven CT, Jan. 1989.

[11] B.R. Fox and K.G. Kempf. Opportunistic scheduling for robotic assembly. In *IEEE Int. Conf. Robotics & Automation*, pages 880–889, St.Louis MO, 1985.

[12] 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.

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

[14] Z. Har'el and R. Kurshan. Software for analytical development of communication protocols. *ATT Technical Journal*, pages 45–59, Jan./Feb. 1990.

[15] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[16] J. Laird. Integrating planning and execution in soar. In J. Hendler, editor, *AAAI Spring Symposium on Planning in uncertain and changing environments*, Stanford CA, Mar. 27–29 1990. Systems Research Center, Univ. of Maryland.

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

[18] D.M. Lyons. Representing and analysing action plans as networks of concurrent processes. *IEEE Trans. Rob. & Aut.*, 9(3), June 1993.

[19] D.M. Lyons and M.A. Arbib. A Formal Model of Computation for Sensory-Based Robotics. *IEEE Trans. Rob. & Aut.*, 5(3):280–293, June 1989.

[20] D. McDermott. Planning reactive behavior: A progress report. In *DARPA Workshop on innovative approaches to Planning, Scheduling and Control*, San Diego, 1990. Morgan Kaufman.

[21] D. McDermott. A reactive plan language. Technical Report YALEU/CSD/RR/#864, Computer Science Dept. Yale University, New Haven CT, August 1991.

[22] D. McDermott. Robot planning. Technical Report YALEU/CSD/RR#861, Yale University, Aug. 1991.

[23] 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.

[24] M. Schoppers. *Representation and Automatic Synthesis of Reaction Plans.* Technical Report UIUCDCS-R-89-1546 (PhD Thesis), Dept of Computer Science, University of Illinois at Urbana-Champaign, 1989.

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

[26] C.J. Sellers and S.Y. Nof. Performance analysis of robotic kitting systems. *Rob. & Comp. Integ. Manuf.*, 6(1):15–24, 1989.

[27] L. Spector and J. Hendler. Knowledge strata: Reactive planning with a multi-level architecture. Technical report UMIACS-TR-90-140, UM Institute of Advanced Computer Studies, Univ. of Maryland, Nov. 1990.

[28] 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.