Achieving Robustness by Casting Planning as Adaptation of a Reactive System.[1]

D. M. Lyons and A. J. Hendriks

Philips Laboratories
North American Philips Corporation
Briarcliff Manor NY 10510

Abstract

Classical AI planning is not sufficiently robust in uncertain and dynamic environments. Reactive approaches are robust in some environments – namely those for which they have been pre-programmed. An approach is presented here that integrates *a priori* planning with reaction. As motivation, a practical robot problem, the kitting robot, is presented. Solving this problem demands a system that can make timely and robust actions in an uncertain environment. A solution to the kitting robot problem is outlined in which planning is cast as adaptation of a reactive system to suit changes in the goals or environment. The reactive system (the *reactor*) is based on a formal model for representing flexible robot plans, the $\mathcal{RS}$ model. Thus, the reactor structure and the mechanisms by which the planner improves the behavior of the reactor can be formalized. This paper concludes by describing how system is being implemented to control a Puma-560 robot equipped with visual sensing.

1 Introduction

Solving the problem of planning and acting robustly in an uncertain environment is a crucial objective in robotics. For example, Fox & Kempf [9] describe how scheduling manufacturing operations suffers from the problem of executional uncertainty: During the execution of a schedule, some of the resources can be expected to become unavailable, but there is no way to tell in advance which one or when. We introduce a practical problem, the kitting robot problem [25], which necessitates producing a robot system that can plan and react in an uncertain environment, and we describe an integrated approach to planning and reacting to address this kind of problem.

Classical AI planning, as exemplified by STRIPS [8] and upon which many planning systems are still based, is not sufficiently robust in uncertain and dynamic environments[22].

[1]An earlier version of this paper was published in the 1991 IEEE Int. Conf. on Robotics & Automation

Reactive approaches, a more recent development exemplified by Brooks' work[5], operate very robustly in some uncertain and dynamic environments — namely those for which the reactions have been pre-programmed. Our solution to the kitting robot problem improves on the state-of-the-art by combining the ability to plan in advance with the ability to react at *all* times to the environment. We cast planning as *adaptation* of a reactive system (the Reactor) to suit changes in goals or the environment. Rather than generating and then executing plans, our approach is to have the Reactor be a real-time system that continually interacts with the environment, and the planner be a separate and concurrent system that 'tunes' the behavior of the reactor to ensure that goals are achieved. The Reactor is based on a formal framework for representing flexible robot plans, the $\mathcal{RS}$ model[14, 16]. Thus, the behavior of the Reactor, and the rules by which the Reactor can be modified, become open to mathematical analysis. We present a set of rules by which the planner 'improves' the reactor to render its behavior more goal-directed. Adaptation of the reactor by the planner is made possible by the unique way in which we represent actions as processes that can communicate with, and be affected by, other actions. The reactor is implemented on a real-time kernel that directly emulates the $\mathcal{RS}$ model [12, 19]. The planner is implemented under the POPLOG system on a SUN3. The planner-reactor system is used to control a Puma-560 robot equipped with two cameras and a force-sensing, multi-fingered gripper.

Section 2 introduces the kitting problem and describes why the current state of the literature does not support solving it. Section 3 introduces the 'planning as adaptation' concept in more detail, and some background information on our action representation, the $\mathcal{RS}$ model. Section 4 describes each component of the system in more detail. Section 5 overviews our implementation environment and presents two short implemented examples.

2 Kitting Robot

A kitting robot is a robot system that puts together assembly kits. Simpler and cheaper automation can construct the assemblies once they have been placed in the kits and routed appropriately. Instead of building a factory full of expensive, intelligent robots, the manufacturer can focus the cost and intelligence into a small number of kitting robots which feed the rest of the factory. Kitting provides the line stock availability, the line scheduling flexibility, and the reduced station cycle time productivity for competitive results [25]. However, to achieve this ideal, the kitting robot has to 'iron out' all the uncertainties associated with the assembly process so that it can be done by simpler automation.

The construction of kits is actually a simple form of assembly: there is much less sequentiality (and therefore, more freedom) in placing parts into kits than in assembly, and mating between part and kit is usually not a high-tolerance procedure. The kitting robot has to deal with the following sources of uncertainty:

1. Variable arrival rates and poses of incoming parts. The primary job of the kitting robot is to take parts (perhaps directly from stores) with high position and pose uncertainty, and to put them in a kit so as to reduce the uncertainty.

2. Errors 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. Substitutability of incoming parts. That is, which parts can be substituted for other, unavailable parts.

4. Mixed batches of kits. The robot may be serving more than one line, or the lines may be running mixed batches.

5. Variable availability of resources. Examples of resources are tools or machines with which the the robot needs to coordinate directly.

6. Response to events in the downstream automation. For example, machines breaking down, or alternate machine configurations.

7. Response to factory management advice. For example, response to changing the batch mix or throughput, or implementing once-off fixes for special situations.

Thus, the key characteristic of the kitting robot is not so much its ability to reason about assembly, but rather its ability to choose timely and effective actions to suit an uncertain and dynamic environment.

In order to deal with events as they happen, the kitting robot needs to be capable of *real-time, reactive response*. On the other hand, in order to deal with changes in the environment (e.g., bursts of errors in parts, or failures of resources) or new goals issued by factory management, the kitting robot needs to be able *to plan ahead 'on the fly'*. The current literature in Robotics and AI does not yet support building a robot control system that successfully incorporates both of these attributes.

Classical AI planners, e.g., STRIPS[8], take one 'snapshot' of the world, and produce a fixed sequence of actions to achieve a goal in that world. This approach is too inflexible to work in an uncertain and dynamic environment: by the time the plan has been produced, the world may well have changed in such a fashion as to render the plan useless. There is an even more important limitation with this approach: STRIPS-like, state-based approaches make it almost impossible to represent *concurrent actions*, and the actions of *external entities* (entities other than the active agent, such as other machines, or even the forces of nature)[22].

More recently, the concept of systems that react, rather than plan, has been introduced by Brooks [5] and Agre & Chapman [1] among others. This approach allows for fast and flexible response to an environment that can be complex, dynamic and uncertain. One example of the advantage of such a reactive system in an industrial setting is contained in Fox & Kempf's paper[9] on opportunistic scheduling.

There are two basic problems with the purely reactive approach: Firstly, to date the building of reactive systems has been largely manual and *ad-hoc*. We argue here that this is inappropriate. A reactive system can be expected to exhibit complex behavior in a complex environment[5]; that is one of the key advantages of the approach. However, human intuition is notoriously unreliable in predicting such complex situations. We should make every effort to use mathematical techniques for predicting and designing reactive behavior if at all possible. This was our motivation in previous work [14, 20]. Bestavros et al.[3] also describe a formal framework for building reactive systems, with many similarities to our approach.

The second problem is that a reactive approach doesn't work well for situations outside the range for which the 'reactions' were programmed[23]. There have been a number of attempts to combine the ability to react with that of *a priori* planning (usually in the context of an assembly workcell) [6, 26]. In general, these systems work by generating a complete plan in advance. This plan is then down-loaded to an Executor or Scheduler, which then chooses an appropriate action sequence based on the state of the environment. This misses an important point however. If a reactive system is indeed called for in the problem specification, then the standard planner-executor division will simply not suffice for two reasons:

1. Generation and loading of entire plans is unrealistic in a dynamic and uncertain environment.

2. An executor (or scheduler) can only act when it has been loaded with a plan; a reactive system needs to be able to act at any time.

In the remainder of this paper, we will describe a novel approach to integrating planning and reaction in the kitting robot domain. The approach has the following characteristics.

1. It is based on a precise framework for reactive plans, the $\mathcal{RS}$ model, and formal techniques for looking at the behavior of plans situated in their environment.

2. It defines planning not as the off-line *generation* of plans, but as the on-line *adaptation* of a reactive system to suit the environment and system goals.

3 Planning and Reacting

The usual definition of a planner is a program that produces a sequence of operations that when applied to the world, will take the world from its initial state to a goal state. Typically,

Figure 1: Planner-Reactor-World System

this plan is then sent to an executer which carries out each of the actions. Our definition of planning is more akin to those of Agre & Chapman [1] or Bresina & Drummond[4]: a planner is a system that continually modifies a reactive system, the reactor[2], so that its behavior becomes more goal directed.

3.1 Overview

The reactor is a system, like those discussed by Brooks and others[3, 5, 15], that contains a network of sensory-motor 'linkages' which we will call reactions. The reactor has several key properties:

1. It can produce action at *any* time. Unlike a classical plan executor, a reactor is always actively inspecting the world, and will act should one of its reactions be triggered. A reactor should produce useful behavior even without a planner.

2. It is a *real-time* system. The central objective of the reactor is to produce the appropriate action at the appropriate time. To make this possible, all the 'intelligence' in a reactor is encoded into reactions and cross-connections between them, and annotated with time-constraints.

The reactor is initialized with a number of generally useful behaviors already in place: distinguishing objects from background, locally avoiding obstacles, scanning its sensors around the environment, etc.

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

[2] Our use of this term echos that of Bresina & Drummond.

planner as a system that interacts with the reactor to produce appropriate behavior, in the same way that the reactor interacts with the world (Figure 1). The input to the planner includes: a description of the environment ($\mathcal{E}$) in which the planner is operating; a description of the reactor ($\mathcal{R}$); and advice ($\mathcal{G}$) from a user about objectives the reactor should achieve (e.g., geometric goals such as kit layouts) and constraints the reactor should obey in its behavior (e.g., batch mix or resource usage constraints).

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

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

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

3.2 The Representation of Actions

The $\mathcal{RS}$ model was developed to express robot plans that could 'tune' their behavior to suit their environment. This is particularly relevant in the robot domain because object locations, identities and behaviors are never precisely known in any realistic task. Plans, actions and world models are all represented as hierarchical networks of distributed processes. For a more detailed motivation and description of $\mathcal{RS}$ see [14, 15, 16]; we present the model here from the perspective of its novel representation of actions.

$\mathcal{RS}$ Terminology. An $\mathcal{RS}$ description of a process, or network of processes, is called a *schema*. For example, $\mathtt{Joint}_v\langle x \rangle$ denotes a process that is an instance of the schema `Joint` with one ingoing parameter v and one outcoming result x. Networks are built by composing processes together using several kinds of *process composition operators*. This allow processes to be ordered in various ways, including concurrently. They also allow 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.

The composition operations are *sequential* `A;B`, *concurrent* `(A,B)`, *conditional* `A:B` and *disabling* `A#B`. The process $\mathtt{T} = \mathtt{P;Q}$ is simply the network of the process `P` executed first, and when that terminates, process `Q` is 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., $(\mathtt{Place}_{part1}, \mathtt{Place}_{part2})$, or parallel actions — actions which need to be done simultaneously, e.g., squeezing an object *obj* with two fingers $f1$ and $f2$: $(\mathtt{ApplyForce}_{f1,obj}, \mathtt{ApplyForce}_{f2,obj})$. Concurrent processes can communicate messages to each other via *communication ports*. Concurrent composition can be augmented with a port-to-port connection mapping c, which indicates which ports are connected together, $(\mathtt{A}, \mathtt{B})^c$.

Conditional composition allows the construction of networks whose behavior is conditional. The network of $\mathtt{T} = \mathtt{P:Q}$ 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. *Disabling* composition allows one process to terminate another. The network $\mathtt{T} = \mathtt{A\#B}$ behaves like `(A,B)` except that it terminates whenever either process terminates.

The final two composition operators are defined recursively in terms of these four: *synchronous recurrent* $\mathtt{A} :; \mathtt{B} = \mathtt{A} : (\mathtt{B}; \mathtt{A} :; \mathtt{B})$, and *asynchronous recurrent* $\mathtt{A} :; \mathtt{B} = \mathtt{A} : (\mathtt{B}, \mathtt{A} :; \mathtt{B})$. Synchronous recurrent composition is similar to *while-loop* iteration. Asynchronous recurrent composition does not iterate, but rather 'spawns' off a set of concurrent processes every time its 'condition' is satisfied.

These operations are summarized in Appendix A.

Plans, Actions and Worlds. The set of basic processes is divided into three classes: basic sensory schemas, basic motor schemas, and basic task schemas. Plan are represented by composing sensory process and motor processes together via task processes. As a point of style, this makes sensing highly task-specific, which allows us to represent the world in a fashion similar to Agre & Chapman's *functional-indexical*[1] terms. That is, objects are only represented in $\mathcal{RS}$ in terms of the role they play in plans.

Another novel feature of this approach is the following. All actions are represented as processes, and can be sent messages and controlled like any other processes. This is very different from the classical view of actions as atomic events that simply 'happen' and are only characterized by preconditions and effects. Our approach allows us to easily handle effects such as modification of the behavior, or 'inhibition', of actions by other actions, that have proven important in building reactive systems [5, 23].

It is not possible to describe the behavior of a reactive system in any concise way without referring to the environment in which the system is active. Therefore, the appropriate unit to analyze is the system embedded or situated in its environment. Our approach in $\mathcal{RS}$ is to build world models from the same tools used to build plans: namely networks of concurrent processes. We demonstrate in [15] that it is possible to construct models of uncertain and dynamic worlds in this fashion, and to analyze the concurrent combination of the world and the reactive system to determine their interactive behavior.

Process Evolution. The analysis of programs in process-based models has been under investigation for some time in Computer Science[10, 11]. We follow the techniques of that field in analyzing plans and worlds in $\mathcal{RS}$. It is important to represent how process networks evolve over time, as processes dynamically terminate or are created. In [14, 15] we introduced the evolves operator, which is defined as follows: We say that process $\mathtt{A}$ evolves into process $\mathtt{B}$ under condition Ω if $\mathtt{A}$ becomes equal to $\mathtt{B}$ when condition Ω occurs; we write this as $\mathtt{A} \xrightarrow{\Omega} \mathtt{B}$. Appendix B summarizes the definition of this operator.

4 The Planner-Reactor System

This section describes each of the components of the planner-reactor in some detail.

4.1 Reactor

The reactor consists of a set of sensory-actions pairs, reactions, and a well-defined interface through which the planner can modify reactor structure. The reactor, and the way in which it can be adapted over time, is specified using the $\mathcal{RS}$ model. The general form of the reactor is the following concurrent network of processes:

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

where

1. $\mathtt{P}^i$ is a reflex (as explained below).
2. $\mathtt{E}^i$, by the definition of #, is a guard process by which (from the definition of #-composition) $\mathtt{P}^i$ can be removed by the planner.
3. `ADDPLAN` is the interface through which the planner can add (from the definition of ::-composition) new guarded reactions $\mathtt{E}^m \# \mathtt{P}^m$.

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

$$\mathtt{Locate}_{p1}\langle obj\rangle \;:;\; (\mathtt{Alloc}_{\chi,R}\langle r\rangle : \mathtt{Place}_{r,obj,dest})$$

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. $\mathtt{Alloc}_{\chi,R}$ allocates a robot[3] according to some constraints χ and only terminates when it gets one, producing a pointer r. `Place` then uses robot r to acquire and move 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 get a robot to acquire and move that part.

Plans versus Reactor Segments. A reactor does not contain/execute plans in the same sense that a classical plan executer/scheduler does, i.e., as a data structure that is manipulated or executed. Nonetheless, certain parts of the reactor structure can be identified as corresponding to the instructions necessary for a specific task or subtask. We need to represent these *reactor segments* in such a way that they can be extended or refined incrementally by the planner (via items 2 and 3 above). We have previously introduced the concept of *situations* in [18] as an intuitive division of the reactions available to the reactor into concurrently enabled sets. Reactor segments are represented in the following *situation-labeled* form.

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

as a concurrent set of actions[4], each of which carries its own monitor process `CheckSit` to determine if the world and reactor present an appropriate situation with respect to task t in which the action (sequence) should be carried out. $sit(t)$ is the set of situations for task t

---

[3]For more details on on-line allocation of robots, see [13].

[4] $\Big|$ is the set notation form of the concurrent composition operator.

for a particular world model. We postpone to the next section a more detailed description of how to get $sit(t)$. Every segment is 'guarded' by a condition monitor `I` that monitors the applicability conditions for the segment; should the monitor fail (`I` aborts), that segment is disabled, and the planner is notified.

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

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

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

$$\mathtt{R} \stackrel{\alpha}{\longrightarrow} \mathtt{R}' \tag{4}$$

where `R'` is the improved reactor, and α is some combination of Ak and Dk conditions. This adaptation mechanism describes the way in which the planner can modify the reactor to improve its behavior.

4.2 Planner

Planning is an ongoing activity. Based on the structure of the reactor, the advice and the perceived environment, the planner continually reassesses the appropriateness of the reactors behavior. If this behavior does not suit the advice anymore, either because the advice has changed or its perception of the environment has changed, the planner considers adapting the reactor.

Important aspects in our approach to planning for a reactive system are: to plan communication between actions; to allow concurrency of actions; and to continually assess the appropriateness of the existing reactor structure and, if necessary, planning incremental adaptations that improve the reactor while at all times maintaining a working reactor structure.

Planning for communicating actions. The action structure we employ, the $\mathcal{RS}$ process, is a much richer structure than the typical action model used in classical planning. In particular, our actions have the ability to communicate with other actions and accept or send parameters from/to other actions. This allows the planner to plan for *knowledge producing* actions, and have the acquired information be sent to another action that needs it. For example, to start a kitting task, the robot needs to find and place a kit tray. The planner cannot assume that the reactor knows *a priori* where all the instances of trays are located in the work area. However the planner knows it has an action at its disposal, `Locate,` which can determine the location of an object instance. Therefore its plan will consist of the `Place` and Locate actions, with an appropriate parameter passing connection between them. In $\mathcal{RS}$ notation this plan is phrased as follows:

$$\texttt{Plan} = \texttt{Locate}_{tray}\langle p\rangle : \texttt{Place}_p$$

The Locate action will locate an instance of the type $tray$ and once found, it will put the information in the communication port p from which the action Place can acquire it.

Another type of communication which has proven important in building reactive systems is communication between concurrently executing actions. For instance, consider the same task as before, but now in a cluttered environment. In that case, it is necessary to use a Safe_Place action, a version of Place which detours around obstacles. To this end, Safe_Place has an input communication port on which it can receive information about the location of obstacles. The process LocalObstacles is continuously active in the reactor (say) and provides information about all objects in the vicinity of the robot. The planner produces the following image of what should be in the reactor:

$$\begin{aligned} \texttt{Plan} &= \texttt{Locate}_{tray}\langle p\rangle \ : \\ &\qquad (\ \texttt{Safe\_Place}_p \ , \ \texttt{LocalObstacles}\ )^c \\ c:&\ \ \texttt{LocalObstacles}(obst) \mapsto \texttt{Safe\_Place}(obst\_info) \end{aligned} \tag{5}$$

In this plan the LocalObstacles action actively looks for local obstacles and as soon as it finds one it puts information about that obstacle on the *obst* port, while continuing to look for others. The Safe_Place action can then retrieve this information from its $obst\_info$ port and use it to carry out local obstacle avoidance.

<u>The Ideal Reactor.</u> The objective of the planner is to incrementally construct a reactor that achieves the planner's goals not only in the current environment, but because change is possible, in all environments that might occur, as indicated by the environment model. A reactor that fulfills this criterion is called an *ideal reactor,* and is similar in behavior to Schoppers universal plan [24]. The behavior of the ideal reactor can be captured with the evolves operator as follows. Let $\texttt{R}^*$ be a process implementing the ideal reactor for environment $\mathcal{E}$ and goals $\mathcal{G}$. Let EM be a process implementing the environment as described by $\mathcal{E}$. The behavior of the reactor in the environment is modelled by the evolution of the concurrent composition $(\texttt{R}^*, \texttt{EM})$. For the ideal reactor, only evolutions of the following form are possible:

$$(\ \texttt{R}^*\ ,\ \texttt{EM}\ ) \longrightarrow (\ \texttt{R}^*\ ,\ \texttt{GEM}\ ) \tag{6}$$

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

<u>Incremental construction of the Ideal Reactor.</u> The ideal reactor is the most robust reactor possible, however, it is likely to be a very large and complex machine. It is unrealistic

to presuppose that the planner will always be able to generate the ideal reactor in one step for the following reasons:

1. It may not have sufficient time, either because there are time constraints within the reactor, or because ongoing change in the environment would cause wasted planning effort.

2. There may be too much uncertainty in the world model for the reactor to be constructed without doing some more sensing.

3. It may not be possible to built the ideal reactor because of resource and action constraints.

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

To investigate this issue, we can restrict the ideal reactor definition above (6) with conditions:

$$(\ \mathtt{R},\ \mathtt{EM}\ ) \xrightarrow{\omega} (\ \mathtt{R},\ \mathtt{GEM}\ ) \tag{7}$$

This says that as long as assumptions ω hold, then $\mathtt{R}$ behaves like the ideal reactor. We call this the ω-ideal reactor $\mathtt{R}^{\omega}$ for environment $\mathcal{E}$ and goals $\mathcal{G}$. As it stands, this claims nothing about the behavior of the reactor if the assumptions don't hold. In the case of failed assumptions, it is necessary that the reactor do nothing that would prohibit a future improved version of the reactor from operating successfully. Note that our reactor segment formulation (2) insists that all reactions are guarded by assumption monitors. Thus when an assumption fails, all the reactions that use this assumption are prevented from interacting with the environment.

In planner terms, ω is a set of assumptions under which $\mathtt{R}$ has been constructed. This entails 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. In parallel, the actual status of the assumptions is examined by perception requests issued by the planner to the reactor.

With this, we can rewrite the evolution of the reactor under planner adaptation (4) as:

$$\begin{gathered} \mathtt{R}^{\omega 1} \xrightarrow{\alpha 1} \mathtt{R}^{\omega 2} \xrightarrow{\alpha 2} \mathtt{R}^{\omega 3} \xrightarrow{\alpha 3} \ldots \xrightarrow{\alpha n} \mathtt{R}^{*} \\ for \quad \omega 1 > \omega 2 > \omega 3 > \ldots > \emptyset \end{gathered} \tag{8}$$

and where αi is some combination of Ak and Dk conditions.

The assumption relaxation priorities are governed by the chosen ordering on ω. Using the likelihood of ω holding in the environment as the ordering criterion enables the planner to build reactor adaptations to deal with the least likely assumptions first. Only then it will take care of increasingly more likely assumptions as it has time. Of course, invariant failures in the reactor or answers from perception requests, which signal assumptions not holding in the world, can change this ordering and thus the relaxation process.

5 Implementation

We have implemented an initial version of a planner-reactor system to conduct kitting robot experiments. Figure 2 shows the basic layout of our implementation. In the next subsection we briefly overview the implementation environment, and in the two subsequent subsections we present two planner-reactor examples built in this implementation.

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

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

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

Figure 2: The Kitting Robot Implementation.

Vision Server. 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 processor functions as a vision server processor and accepts two kinds of commands: A `recognize` command causes it to scan the image from one of the cameras and determine if there are any instances of a given set of 2D object models in the image. If there are, then the template name, the 3-D position and the angle in the worksurface plane are returned for each one. The pose restriction mentioned previously comes about because of the limited set of object models. The second command is the `track` command. This causes the vision server to issue commands to the robot server so as to match the position and velocity of the object currently seen by the local camera.

$\mathcal{RS}$ System. A subset of the $\mathcal{RS}$ model has been implemented as a robot programming language. The `RS-L3` programming environment consists of a YACC/LEX-based parser and a real-time executer. The parser accepts systems of process definition equations in a computer-keyboard version of the $\mathcal{RS}$ syntax. It passes on these parsed definitions, as they are made, to the real-time executer, which executes them as dictated by the model semantics. Thus, all execution is interpretive. The executer implements composition, port-i/o, buffer management, and scheduling. It uses threaded-execution to implement concurrency. The thread scheduler implements an *earliest deadline first* real-time scheduler[5]. The parser always

[5]For a description of our progress to date in real-time constraints, see Lyons & Hendriks (1991)[17].

runs on a SUN and communicates, using UNIX sockets, with a remote executer. The executer can be run on one of the VMEBUS processors or on a SUN. For the example described in the next section, the system was run on a SUN (for memory reasons).

A set of basic sensory and motor schemas has been built to interface to the robot and vision servers. These basic schemas run as atomic processes that send command messages to the servers and return results from them. For example, $\texttt{Locate}_m$ is a basic sensory process that continually queries the vision server with `recognize(m)` commands, and only terminates when such a model instance is seen. $\texttt{Domove}_p$ is a basic motor process that transmits a `move(p)` command to the robot server, and terminates when that command has been carried out.

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

Planner. The planner uses Pelavin's[22] Interval Temporal Logic (ITL) formalism to represent world models. It maintains these models as sets of temporal predicates and rules. Temporal relations between intervals of time are maintained in a network and new relations are enforced by constraint propagation [2]. Formulas in ITL are first order logic predicates augmented with a modal operator @, e.g., $IN(A,D)@i1$ specifies that the predicate $IN(A,D)$ should be evaluated over interval $i1$. In addition the special predicate $ITL(interval1, itl-relation, interval2)$ is used to specify constraints between intervals, e.g., $ITL(i1, equal, i2)$ specifies that the interval $i1$ should denote the same interval of time as $i2$. Other relations we use here include $finishes$ (both finish at the same point) and $encompasses$ (the second is a subinterval of the first).

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

5.2 Example One

Consider a kitting robot which has the following as part of its current reactor structure:

$$\texttt{P} \quad = \quad \texttt{Locate}_A\langle r\rangle :; (\texttt{Perceive}_r \ , \ \texttt{Place}_{r,buf}) \tag{9}$$

That is, it will look out for any objects in the set A, and each one it finds, it will place into the buffer area buf. The process `Perceive` updates the planners knowledge about the world to include the fact that a new instance of A has arrived. Let `Place` have a port inh which can be used to inhibit the process from taking any action. For convenience, let us also assume there is a process `Inhibit` that produces 'inhibit' signals, and can be connected to the inh port to inhibit `Place`.

The planner has a model of the purpose of the reactor structure. The ITL representation is as follows:

$$\forall x, i1, i2: \quad Instance(x, A)@i1 \Rightarrow Available(x)@i2 \wedge ITL(i2, equal, i1)$$

This specifies that the reactor maintains availability of an object by placing it in the buffer area.

Now consider the case where the system is given the extra task of responding to the requests of downstream automation for objects in the set $T \subset A$, and with placing instances of those objects on the transport belt. It receives the following additional goal:

$$\forall x, i1, i2: \quad (Instance(x, T) \wedge Request(T))@i1 \Rightarrow Onbelt(x)@i2 \wedge ITL(i1, encompasses, i2)$$

This new goal requires the planner to modify the reactor structure. First, the occurrence of a request has to be determined, and conditional on this the existence of an appropriate object instance, action should be taken to locate and put that instance on the belt. However, since $T \subset A$, this plan of action could interfere with the structure already in place in the reactor, resulting in an attempt to place the same instance both on the belt and in the buffer. Since placing the instance on the belt also maintains availability to the system, an inhibit of the place-in-buffer action resolves the potential interference. After the modifications have been sent to the reactor, the new reactor structure is

$$\begin{array}{rcl} \mathtt{P} & = & \mathtt{Locate}_A\langle r\rangle \ ::\ (\ \mathtt{Perceive}_r\ ,\ \mathtt{Place}_{r,buf}\ , \\ & & \quad \mathtt{Request}\langle T\rangle\ :\ \mathtt{IsType}_{r,T}\ : \\ & & \quad\quad (\ \mathtt{Inhibit}\ \#\ \mathtt{Put}_{r,Belt}\ )\ )^c \\ & c: & \mathtt{Inhibit}(inh) \mapsto \mathtt{Place}(inh) \end{array} \tag{10}$$

5.3 Example Two

Consider the following kitting robot example: The robot looks for a kit tray t, fills in parts $p1, p2$ and $p3$ (which arrive via a conveyer belt) on that tray, and then sends the tray out of the cell. We will use this example to illustrate how the planner determines what changes to make to the reactor in the case i) that it is incrementally constructing the reactor, and ii) that it is adapting the reactor to suit a perceived change in the environment.

New Goals. The goal in this example is

$$\forall t, i1 Instance(tray, t)@i1 \Rightarrow \exists i2 Filled(t)@i2 \wedge ITL(i2, finishes, time) \tag{11}$$

Figure 3: Kitting Example Situations.

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

$$Filled(t)@i1 \iff$$
$$[\ \exists x, y, z Instance(p1, x) \wedge Instance(p2, y) \wedge Instance(p3, z) \wedge Contains(t, x, y, z) \wedge AT(t, buffer)\ ]@i1 \tag{12}$$

That is, that t contains an instance of each of the part types $p1, p2$ and $p3$ and that it is has been sent to the output buffer of the cell.

Problem Solving Context. Each goal is indexed into the domain theory. Invoking the goal results in certain actions, assumption descriptions and general domain rules being accumulated in the problem solving context, PSC. For example, *Instance* brings in potentially useful actions such as `Locate` and `Acquire` ; *Contains* brings in a number of subgoals, such as $In(t, x)$. Assumptions are predicates that are accumulated into the PSC, and about which there exists a domain rule stating their likeliness. In this example, the only assumption is brought in by `Acquire`. A precondition for $\mathtt{Acquire}_x$ is that we have $static(x)$, i.e., that x not be a moving object. We have a domain rule in this case that says $static(x)$ is likely. Thus, although the planner does not know $static(x)$, it asserts it in the current PSC, and also it into the set of assumptions.

Regressing the goal $Filled(t)$ in the PSC gives us the first abstract description of the reactor (figure 3): first locate any tray, then fill that tray, and finally remove the tray to the output buffer. This expectation is decomposed to yield the following reactor segment:

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

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

<u>Refinement.</u> On the next refinement the $tray$ situation is decomposed, to extend the situations that the reactor can deal with. Given that the parts can be placed in any order, this results in the following reactor segment:

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

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

The addition of the final situation segment $Filled$ is straightforward.

<u>Forced Refinement.</u> Consider now the case one of the assumption monitors fails, indicating that the robot has interacted with a moving object. Say this occurred on part $P2$. That reaction is immediately disabled by the failure of its $\mathtt{I}_{static}$ monitor, and the planner notified. The planner now removes $static(x)$ from its assumption set, and sets it's value to false in the PSC (since it now knows its value). Since the set of assumptions has been changed, a new expectation is generated. For this example we choose a conservative approach, and regenerated only segment directly violated (though the assumption should then be propagated to the whole reactor). Thus, the expectation now again contains the situation $LacksP2$. In our example, there is another action `Track` which allows the robot to acquire a moving object. Decomposing the expectation in this updated PSC produces the new reflex:

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

The reactor is updated by commanding $\mathtt{E}_5$ to terminate, and commanding `ADDPLAN` to terminate with $k = 7$.

6 Discussion

Summary. We have presented the kitting robot problem, a practical motivation for building robot systems that can plan and act robustly in an uncertain and dynamic environment. We have discussed why the state of the literature does not yet support building such a system. Our work attacks this problem by casting planning not as an *a priori* activity, but as a continuing adaptation of a reactive system. We use a novel action representation, the $\mathcal{RS}$ model, to facilitate the on-going adaptation of the reactor. The laws by which the planner improves the behavior of the reactor were presented formally. The planner is also unique in that it plans for concurrent communicating actions.

Comparisons. Our approach to the problem of planning and reacting is centered on the concept of having separate, but coupled, planner and reactor systems. It is possible to combine planning and action in a single architecture. Robo-Soar [**?**] for example uses a single component for both planning and acting. While this has tremendous advantage in allowing for arbitrary interaction between planning and execution, it also has the disadvantage that it allows for arbitrary interference between execution and planning. For example, when an event that should be reacted to occurs, Robo-Soar loses its intermediate planning state in order to react to the event. In anthropomorphic terms, Robo-Soar lacks the ability to engage in non-conscious, routine activity. Our approach maintains a tight coupling between planning and reacting, but keeps them separate to avoid interference, and thus provides for a level of 'non-conscious', but controllable, behavior.

The most directly related work to ours is that of Bresina & Drummond [4, 7]. They describe an architecture, ERE, that integrates planning, scheduling and control. Their reactor is based on if-then rules and has the flavor of a real-time expert system. Our reactor, based on process-networks, has more of the flavor of a compiled controller. One major difference is that our richer model of actions provides for a way to discuss actions modifying or inhibiting each other while they are in progress. Rather than having conditional actions, we have conditional networks of actions.

Bresina & Drummond organize their architecture around their *principle of independent activity.* Our argument is that they haven't gone far enough with this, and our reactor and planner are less tightly coupled than their components — the planner interacts with the reactor after the style that the reactor interacts with the world. The planner doesn't have available the direct sensory impressions nor internal state of the reactor, only that data that are echoed back as perceptions. (Of course the planner gets to say what perceptions are returned). This simplifies the planners world model: it doesn't have to consider sensory data that it knows the reactor is primed to take care of, and it can have the reactor monitor specific conditions (such as action invariants, subgoals, opportunities, etc) and signal back to redirect its reasoning should one of these occur. Thus, even as the planner facilitates the reactor, by allowing it to be 'myopic', the reactor facilitates the planner, by allowing it to ignore some of the multitude of sensory data constantly available. This could be called the *principle of complementary activity.*

Nilsson [21] has described an action network structure for *teleo-reactive* agents — agents that take changing environmental and goal conditions into account in acting. His ASTRAL language allows a user to specify control networks, networks of combinatorial circuits, for intelligent agents. The 'planning' component of a network is 'built-in' in the sense that goals are hard-wired to fire particular actions. This representation is similar in effect to Shoppers' [24] universal plans, though Nilsson does not demand his networks be universal. The appropriate point of comparison between our work and Nilsson's is our reactor to his action networks. We would maintain that all the ingredients of an action network, including precompiled goal to action linkages, are appropriate reactor material. A crucial point of similarity is that our process model and Nilsson's 'circuit' model both address the issue of 'communications' between actions *while* the actions are in progress. Our work additionally specifies how a specific 'action network' should be updated over time to reflect long-term changes in the environment.

Status & Future Work. The current status of this work is as follows. The planner and reactor have been tested on the robot workcell for the case where the planner adds in goal-directed actions to an existing reactor. On-going adaptation of the reactor by the planner has been tested for single adaptations, but not for iterated adaptation. Further theoretical development is necessary, for example, to ensure that the planner-reactor relationship is stable and efficient, and to determine what the appropriate 'units' are by which to incrementally adapt plans. Our initial research towards answering these questions indicates that we have to extend our representation to include probabilistic information. The existing $\mathcal{RS}$ implementation supports real-time scheduling, but, as yet, we have not experimented with time-constrained plans in reacting or planning.

References

[1] P. Agre and D. Chapman. From reaction to participation. In *DARPA Planning workshop*, pages 123–154, Santa Cruz, CA, Oct. 1987.

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

[3] A. Bestravos, J. Clark, and N. Ferrier. Management of sensori-motor activity in mobile robots. In *IEEE Int. Conf. Robotics & Automation*, pages 592–597, Cincinnati, OH, May 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] R. Brooks. A robust layered control system for a mobile robot. *IEEE J. Rob. & Aut.*, RA-2(1):14–22, Mar. 1986.

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

[7] M. Drummond and J. Bresina. Anytime synthetic projection: maximizing the probability of goal satisfaction. In *AAAI-90*, pages 138–144, Jul. 29th – Aug. 3rd 1990.

[8] 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.

[9] B.R. Fox and K.G. Kempf. Reasoning about opportunistic schedules. In *IEEE Int. Conf. Robotics & Automation*, pages 1876–1882, Raleigh, NC, 1987.

[10] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

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

[12] D. Lyons, S. Mehta, and P. Gopinath. Robust representation and execution of robot plans. In *EuroMicro Real-Time Systems Workshop*, Horsholm, Denmark, June 1990. (Philips TN-89-097).

[13] D.M. Lyons. On-line allocation of robot resources to task plans. In *SPIE Symp. on Advances in Intelligent Robotic Systems; Expert Robots for Industrial Use (Vol. 1008)*, pages 215–222, Nov. 1988.

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

[15] D.M. Lyons. A formal model for reactive robot plans. In *2nd International Conference on Computer Integrated Manufacturing*, RPI, Troy, NY, May 21–23 1990.

[16] 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.

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

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

[19] D.M. Lyons and S. Mehta. A distributed computing environment for the multiple robot domain. In *Fourth International Conference on CAD, CAM, Robotics and Factories of the Future*, New Delhi, India, Dec. 19–22 1989.

[20] D.M. Lyons, R. Pelavin, A. Hendriks, and P Benjamin. $\mathcal{RS}$: A formal model for reactive robot plans. 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.

[21] N.J. Nilsson. Action networks. In J. Weber, J. Tenenberg, and J. Allen, editors, *From Formal Systems to Practical Systems*, pages 21–52. Dept. of Computer Science, Univ. of Rochester, Oct. 27–29th 1988.

[22] 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.

[23] J. Sanborn and J. Hendler. A model of reaction for planning in dynamic environments. *AI in Engineering*, 3(2):95–102, 1988.

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

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

[26] 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.

A $\mathcal{RS}$ Composition Operators

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

B Process Evolution

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

$\mathtt{STOP}$ is a process that immediately terminates. $\mathtt{ABORT}$ is a process that immediately aborts.