Safely Adapting a Hierarchical 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

This paper builds upon work in developing a theory of planning as adaptation of a reactive system. In previous work[7, 6], we introduced the planner-reactor architecture and investigated the structure of the reactor and the planner. In this paper we present our results in representing hierarchical reactive systems and in making safe structural adaptations to an ongoing reactive system. This work is done in the context of automated robotic kitting for assembly and the examples in this paper are taken from this area.

Although hierarchical representation is essential for complex tasks, it poses the problem that it may destroy the reactiveness of the system. We demonstrate that it is possible to choose a 'flat' representation that both preserves reactivity and allows hierarchical structuring. Previous work on modifying ongoing reactive systems has not addressed the crucial issue of making the modifications in a safe and consistent manner. We present a set of desirable properties for an adaptation mechanism: safeness (reactions aren't interrupted), consistency (use only the old reactions or only the new ones) and boundedness (the changes will eventually occur). We then introduce an adaptation mechanism that has these properties.

SPIE Symposium on Intelligent Robots and Computer Vision XII, Boston MA 1993.

1 Introduction

A robot system operating in an environment in which there is uncertainty and change needs to combine the ability to react with the ability to plan[9, 4, 1]. We have previously introduced a practical problem, the kitting robot problem[7], that necessitates integrating *reactive, real-time* responses with the ability to *plan 'on the fly'* and we briefly review this problem here. In previous papers we proposed a solution to the problem of integrating planning and reaction: cast planning as the continual structural *adaptation* of an ongoing reactive system. In (Lyons & Hendriks 1991)[7] we introduced the planner-reactor architecture, a robot control architecture based on this approach, and discussed its application in the kitting robot domain. In (Lyons & Hendriks 1992)[6] we investigated the issue of iterative adaptation of a reactive system by a planner. In this paper we present our results in representing hierarchical reactive systems, and in making safe structural adaptations to an ongoing reactive system.

Previous work on modifying a running reactive system[1, 8, 2] does not address the practical issues of safely removing or disabling some reactions and adding or enabling new ones. If the reaction being removed or added is allowed to execute during this modification then the behavior of the system becomes quite unpredictable. This may result in undesired side-effects on the environment.

We present a set of desirable properties for an adaptation mechanism: safeness (reactions aren't interrupted), consistency (use only the old reactions or only the new ones) and boundedness (the changes will eventually occur), and we introduce an adaptation mechanism that has these properties. Implementing safe adaptation for this problem has entailed some extensions to our previous work. First, to address kitting problems of realistic size, it becomes important to be able to represent a reactor containing hierarchically nested reactions. This is not a trivial change, since adding into a hierarchical structure may destroy the 'reactiveness' of the system due to the depth of the hierarchy. This paper motivates our choice of representation for hierarchical reactions. Second, the adaptation vocabulary of (Lyons & Hendriks 1992)[6] had to be updated to include certain semaphore-like operations to allow for atomic modification actions.

A kitting robot is a robot system that puts together assembly kits — trays containing all the necessary parts from which to build a specific product. Simpler and cheaper automation can construct the assemblies once they have been placed in the kits and routed appropriately. 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 changing environment. The kinds of dynamic and uncertain events the kitting robot needs to handle are described in more detail in (Lyons & Hendriks 1991)[7]. To handle events as they happen, the kitting robot needs to be capable of *real-time, reactive response.* On the other hand, to handle 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'.*

In our previous work[7, 6] the kitting robot examples considered were kept relatively simple. This sufficed to flesh out the theory of planning as adaptation, but it is not sufficient to allow us to examine the behavior of such a system in practice. This paper describes a kitting robot example designed to allow us to look at the practical performance of a planner-reactor system. The example contains 10 assumptions that the planner can use to simplify interaction with the environment. The initial reactor demands 10 situations, each with one or more reactions. The completely adapted reactor demands 22 situations, each again with one or more reactions.

The next section reviews our planner-reactor work, as background for the example and the extensions. The new kitting example is presented in section 3 to motivate the extensions. The hierarchical representation is presented in sections 4 and 5. The new adaptation mechanism is presented in section 6.

2 The Planner-Reactor Approach

In our approach, a planner is not a system that, in response to goals, produces plans to be carried out by a plan executer. Instead, 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*[*]. Figure **??** illustrates our architecture: The reactor interacts directly with the uncertain and changing environment. The planner is a separate and concurrent system that monitors the reactor and its environment by issuing perception requests. When, in the planner's view, the reactor structure will not produce behavior that conforms to the planner's goals, the planner issues adaptations, structural changes to the reactor, to bring reactor behavior back into line with the goals. For a fuller introduction to the planner-reactor approach see (Lyons & Hendriks 1991,1992)[7, 6]. We model the reactor, both formally and in implementation, as a network of concurrent, communicating processes using the $\mathcal{RS}$ model[5, ?].

[*] After terminology established by Bresina & Drummond.

2.1 The $\mathcal{RS}$ Model

Plans, actions and world models are all represented in $\mathcal{RS}$ as hierarchical networks of concurrent processes. The $\mathcal{RS}$ language allows a user to construct programs by 'gluing' together processes in various ways. This style of language is called a *process-algebra* language. These languages have the additional advantage that they facilitate the mathematical analysis of programs. The $\mathcal{RS}$ model was developed to allow users to represent and analyze the kind of programs involved in sensory-based robotics.

A 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 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. The main composition operations are *sequential* `A;B`, *concurrent* $\mathtt{A}|\mathtt{B}$, *conditional* `A:B` and *disabling* `A#B` and two recurrent operators *synchronous recurrent* $\mathtt{A} :; \mathtt{B} = \mathtt{A} : (\mathtt{B}; (\mathtt{A} :; \mathtt{B}))$, and *asynchronous recurrent* $\mathtt{A} :: \mathtt{B} = \mathtt{A} : (\mathtt{B} \mid (\mathtt{A} :: \mathtt{B}))$. (See the Appendix for a description of the composition operators.)

It is important to represent how process networks evolve over time, as processes dynamically terminate or are created. This is captured by the *evolves* operator: We say that process `A` evolves into process `B` under condition Ω if `A` possibly becomes equal to `B` when condition Ω occurs; we write this as $\mathtt{A} \xrightarrow{\Omega} \mathtt{B}$. A process *must* evolve to one of the set of processes to which it can possibly evolve.

2.2 The Reactor

The fundamental structuring concept used in the reactor is a general concept we refer to as *reactor situations*[6]. Strictly speaking, situations are equivalence relations on the state-space for a specific task in a specific environment. Multiple situations may be active at one time. For the reactor, a *situation* being *active* expresses the appropriateness of a particular set of reactions; those *associated with* that situation. Situations have a number of useful characteristics for representing a reactor, see (Lyons & Hendriks 1992) [6] for details.

The following $\mathcal{RS}$ definitions describe the reactor as a set of guarded sensory-action pairs (reactions) and a well-defined interface through which the planner can modify the reactor structure:

$$\mathtt{REACTOR} = \Big(\Big|_{Sit} (\mathtt{E}^i \# \mathtt{P}^i) \;\mid\; \mathtt{ADD}\langle k\rangle :: (\mathtt{E}^k \# \mathtt{P}^k) \Big) \tag{1}$$

where there are three main items:

1. $\mathtt{P}^i$ is a reaction or reactor segment (as explained later).
2. $\mathtt{E}^i$ is a guard process by which the planner can remove (by the #-composition) $\mathtt{P}^i$ from the reactor.
3. `ADD` is the interface through which the planner can add (by the ::-composition) new guarded reactions $\mathtt{E}^k \# \mathtt{P}^k$.

A reactor does not contain/execute plans in the same sense that a classical plan executor/scheduler does, i.e., as a data structure that is manipulated or executed. The reactions are represented procedurally, so the reactor is more like a compiled controller. Nonetheless, certain parts of the reactor structure can be identified as corresponding to the instructions necessary for a specific task or subtask; we call these parts *reactor segments.* They consist of a set of reactor situations and the reactions associated with them:

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

In this definition, $\mathtt{SIT}_s$ is a basic process (see Table 2) that suspends itself until situation s has been asserted. Whenever $\mathtt{SIT}_s$ terminates, it enables (via the :; operation) the reactions associated situation s. $sit(t)$ is the set of situations for task t for a particular world model. Every segment is 'guarded' by a condition monitor I that monitors the applicability conditions for the segment; should the monitor fail (I terminates), that segment is disabled (by the #-composition). An I process transmits a message back to the planner on termination, a perception, that indicates a change in the environment.

In (Lyons & Hendriks 1992)[6], we show that it is possible to write down how the reactor is affected by the changes described in items 2 and 3 of eq. (1) as follows:

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

Saying that a process P successfully terminates (under some condition) is the same as saying that P evolves to STOP (under that condition). So in (3) 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:

$$\mathtt{R} \xrightarrow{\alpha} \mathtt{R}' \tag{4}$$

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

2.3 Incremental Reactor Adaptation

The ultimate objective of the planner is the construction of a reactor that achieves its 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 plans[9]. It is unrealistic to suppose that a planner will always be able to generate the ideal reactor in one step, and in (Lyons & Hendriks 1992)[6] we introduced the more restricted ω-ideal reactor $\mathtt{R}^\omega$. In planner terms, ω is a set of *assumptions* under which the reactor $\mathtt{R}^\omega$ has been constructed. This suggests the following incremental reactor construction strategy: ω is initially chosen to allow the planner to quickly produce a working reactor and then ω is gradually relaxed over time. The evolution in (4) can be used to represent this strategy as follows:

$$\begin{array}{l} \mathtt{R}^{\omega 1} \xrightarrow{\alpha 1} \mathtt{R}^{\omega 2} \xrightarrow{\alpha 2} \mathtt{R}^{\omega 3} \xrightarrow{\alpha 3} \ldots \xrightarrow{\alpha n} \mathtt{R}^* \\ for \quad e(\omega 1) > e(\omega 2) > e(\omega 3) > \ldots > e(\emptyset) \end{array} \tag{5}$$

where the αi are the adaptation commands generated by the planner and $e()$ is an evaluation function. The assumption relaxation priorities are governed by the ordering on ω dictated $e()$. Using the likelihood of ω holding in the environment as the ordering criterion enables the planner to build reactor adaptations to abolish the least likely assumptions first.

The conditions that trigger the planner to produce an improved reactor are described in (Lyons & Hendriks 1992)[6] and the constraints on convergence to $\mathtt{R}^*$ overviewed. These constraints are that, firstly, there must be finite progress, as defined by $e()$, in each improved reactor. Secondly, it must be possible to get to the ideal reactor by relaxing assumptions in any order (since we have no control over what order the environment may choose); there should be no 'dead-ends.'

3 The Extended Kitting Task

The kitting task described here is an extension of the simple task we have described in previous work[7, 6] 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.

The purpose of the kitting robot is to produce kits and filter 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 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 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.

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, their likelihood of holding, and methods to detect whether they hold or not. In the case that the environment goes through regimes or intervals of time over which different subsets of assumptions hold, the planner can capitalize on this, especially when new factory goals have been received and resultant changes in behavior 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. As the environment goes through regimes, the planner can utilize this information to maintain a minimal and efficient reactor.

The set of assumptions for the extended kitting environment is shown in Table 1 in reverse order of likelihood of holding; i.e., in the order in which the planner considers relaxing them. The assumptions are referred to by acronyms such as AS (Assumption of Substitutability), AQ (Assumption of Quality) and so on. The 'Options' column indicates how many ways there are to relax an assumption. The simplest case is a binary assumption. Consider for example the assumption of quality AQ. Our table specifies only two options for this assumption – (Y) it holds, and all parts are of good quality, or (N) it doesn't hold and quality checking will need to be done on all parts. It is possible to consider extending this to more than two options: for example, allowing a statistical quality checking option between the two extremes. In that case, there would be three options for AQ. In the case of the assumption of substitutability AS, the number of options are, for each part p, either no substitutability, or the set of substitutable variants for that part N_p.

	Assm	Description	Options
1.	AS (substitutability)	Whether parts can be substituted for each other.	N/N_p
2.	AQ (quality)	Whether parts can be of variable quality.	Y/N
3.	AF (filled tray)	Whether trays arrive partially filled.	Y/N
4.	ACT (competition)	Whether parts put in a tray will stay there.	Y/N
5.	ACP (cooperation)	Whether trays can be finished by some other station.	Y/N
6.	AM (motion)	Whether objects move around once delivered on the belt.	Y/N
7.	ADT (tray disturbance)	Whether trays remain where they were put.	Y/N
8.	AP (parts homogeneity)	Whether the parts arrive evenly distributed.	Y/N
X.	ADD (downstream disturbance)	Whether downstream automation is always ready to receive finished kits.	Y/N
X.	AO (ordering)	Whether there are ordering constraints between the parts.	N_o

Table 1: The Set of Assumptions for the Extended Kitting Task.

The last two assumptions in Table 1, ADD and AO, cannot be relaxed iteratively like the others. Parts ordering, AO, is the assumption that the parts can be placed in the kit in a particular ordering. The planner might arrive at that ordering by geometric reasoning. The ordering may be partial, to allow for some run-time optimization[3], or the ordering might fail in practice in some regimes due to unmodeled phenomena. The AO options give the kitting robot the ability to modify parts ordering should that be warranted. This modification is treated as relaxation of the AO assumption. There are N_o relaxations of AO, where N_o is the number of parts reorderings allowed at the kitting station.

Whether downstream can accept more kits or not (ADD) is also something that the planner does not relax iteratively. If downstream automation signals a problem then ADD is withdrawn. It is reinstated when downstream automation signals that it is again willing to accept kits. The kitting robot behavior when ADD is withdrawn may be as simple as signalling its upstream stations to stop (e.g., in a strict flow-line factory), or it may involve some amount of in-line buffering. The latter has the advantage of 'smoothing out' delays caused by interruptions in the line, but has the disadvantage of building up stock on the factory floor.

In addition to its normal relaxation sequence, the planner can also be forced to relax assumptions. The planner's choice of relaxation sequence will be altered depending on what assumptions hold when carrying out the task in practice. For example, the AP assumption is the last one the planner would normally consider relaxing; nonetheless, should this assumption not hold in the initial environment, then the planner has no choice but to relax it. Previous results[6] have demonstrated that for the planner-reactor system to converge on any desired reactor, it must be possible to withdraw the assumptions in any order.

Whenever the planner builds a reactor that depends upon a particular assumption holding, it needs to embed a monitor process that can detect whether the assumption actually holds in practice or not. The monitor process triggers a 'clean up' action if the assumption ever fails, and notifies the planner that it needs to relax this assumption next.

Figure 1: Kitting Initial Situation Configuration.

4 The Kitting Task Situation Hierarchy

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, the set of reactions that should be enabled in that situation. Situations can be hierarchical in that reactions for one situation may assert a second. The situation hierarchy for the initial reactor in the kitting task is shown in Figure 1. This is essentially the same as the reactor structure for the simplified kitting task we presented in (Lyons & Hendriks 1992)[6].

The situation hierarchy figures are read as follows. The topmost line is the highest level situation (e.g., BUILDKITS for the initial reactor). The line underneath this indicates how this situation is broken down hierarchically. For example, in the initial reactor, BUILDKITS is broken down into a sequential composition of first FINDTRAY and then HANDLETRAY. Figure 1 shows that HANDLETRAY in turn is composed of three sequential situations, GETTRAY, PLACEPARTS and REMOVETRAY. PLACEPARTS is then composed of three concurrent situations HANDLE-PART-A, HANDLE-PART-B and HANDLE-PART-C. The lowest levels of the hierarchy, e.g., FINDTRAY, GETTRAY, HANDLE-PART-A, and so on, contain reactions. Recall that activating a situation means that a particular set of reactions now become enabled. In FINDTRAY, these are the reactions to bring a tray into the workspace; in HANDLE-PART-X they are the reactions to place part-X into the tray. The actual behavior of the robot in a situation will simply depend on what choices the environment offers.

As each assumption in Table 1 is relaxed, the situation hierarchy is extended. Figure 2 shows the final state of the situation hierarchy. The additional situations are tagged with the assumptions that cause them to be introduced. For example, initially the HANDLE-PART-X situation is atomic. That means its reactions do not assert any other situations, they directly carry out sensing and motion. (For more detail, see our previous examples[7, 6].) However, if the substitutability assumption (AS) is relaxed, then HANDLE-X is expanded out into FIND and GETPUT. The FIND situation can now be given the additional reactions to handle substitutability while the GETPUT situation handles acquiring and placing the object. If the motion assumption (AM) is then relaxed, GETPUT is split up in turn into GET and PUT, where GET now contains a tracking reaction to handle moving parts.

The convergence constraint demands that we be able to make assumption relaxation adaptations to the situation hierarchy in any order. This enforces a commutativity property as shown in Figure 3. This figure

Figure 2: Kitting Final Situation Configuration.

shows how the same situation hierarchy is produced no matter whether the planner relaxes assumption AM and then AS (the left-hand path), or AS and then AM (the right hand path).

5 Representing Hierarchical Situations

Our reactor structure in (1) does not provide for hierarchical representation and thus needs to be extended. Hierarchical representation is essential for complex tasks. The difficulty in providing hierarchical representation for reactions is that there is a danger of losing 'reactiveness' 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:

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

Figure 3: Commutativity of Adaptations.

The reactor is a set of concurrent processes as described in (1): `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* hiearchy. 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:

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

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.

6 Safely Adapting a Hierarchical Reactor

The adaptation equations (3) 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

- $\texttt{ASSERT}_{s,p,\ldots}$ 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.
- $\texttt{SIT}_s\langle k, p, \ldots\rangle$ Terminates if an instance of situation s is currently asserted. k is the instance number and $p, \ldots$ are the current values of the situation parameters.
- $\texttt{WAIT}_s$ Terminates when no instance of situation s is active.
- $\texttt{KILL}_n$ Causes the process $\texttt{E}_n$ to terminate.
- $\texttt{INHIBIT}_s$ Prevents any reaction of situation s from being enabled.
- $\texttt{UNINHIBIT}_s$ Removes the inhibition on situation s.
- $\texttt{WAITINHIBIT}_s$ An atomic combination of `WAIT` and `INHIBIT` on situation s.

Table 2: Situation Control basic processes.

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. (This also applies to the relationship between the assumption monitor process (`I`) and a reaction.) New reactions may be have to be started all 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 has these properties. We will assume the situation control processes shown in Table 2.

Now 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.`

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

An adaptation event is initiated and controlled by the planner using the basic reactor adaptation equations (3). 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. In order to guarantee the boundedness property it is thus necessary to assume that all situations terminate. Furthermore, the numeric bound on adaptation is 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 it 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 $\texttt{P}m$. 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. This adaptation sequence is shown in below:

Step 1 $\texttt{WAITINHIBIT}_{sa}$

Step 2 $\texttt{KILL}_2;\texttt{KILL}_3$

Step 3 Define `P4, P5, P6`

Step 4 `ADD` terminates with $m = 6, 5, 4$

Step 5 $\texttt{UNINHIBIT}_s$

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 examples, `INHIBIT` can be used to 'freeze' a running situation. However, these may require the planner to deliberate about cleaning up after the interrupted situation.

7 Discussion

Addressing more complicated and realistic versions of the automated robotic kitting problem has forced us to extend our work in planning for adaptation of a reactive system to include the issues of representing hierarchical reactive systems and of making safe and consistent adaptations. The paper demonstrates that it is possible to represent hierarchical reactive systems in such a fashion that the hierarchy doesn't compromise the reactiveness. Making adaptations safely to an ongoing reactive system has been sidestepped in the literature to date. This paper presents an adaptation mechanism that is safe, consistent and bounded.

We would expect that a real-world example might contain an order of magnitude more assumptions, situations and reactions than this. Nonetheless this example has the advantage that much of it can be worked out by hand for comparison purposes.

In this and previous work we have assumed a *monotonic decreasing* adaptation strategy; i.e.,, the number of assumptions used in the reactor decrease monotonically in (5) until the ideal reactor is constructed. However, it is clear that some assumptions (e.g., ADD) may need to be repeatedly withdrawn and reintroduced as the environment goes through regimes (defined as intervals of time over which a given set of assumptions holds). The current theory does not support reintroducing assumptions and should be extended to understand how convergence to the ideal reactor for a regime relates to the monotonic strategy.

References

[1] 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.

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

[3] L.S. Homem de Mello and A.C. Sanderson. Task sequence planning for assembly. In *12th IMACS World Congress on Scientific Computation*, Paris, France, Jul. 18-22 1988.

[4] L.P. Kaelbling. Goals as parallel program specifications. In *AAAI-88*, pages 60–65, 1988.

[5] 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.

[6] D.M. Lyons and A.J. Hendriks. Planning for reactive robot behavior. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1992. (See also video proceedings).

[7] 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.

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

[9] 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.

Appendix A. The $\mathcal{RS}$ Composition and Evolves Operators

1. Concurrent Composition: $\mathtt{T} = (\mathtt{P} \mid \mathtt{Q})^c$. T behaves like P and Q in parallel, connected by port-to-port mapping c. The mapping is omitted if empty. This can also be written as $\mathtt{T} = \Big|_{i \in I} \mathtt{P}^i$. This operator is *idempotent*: $\mathtt{P} \mid \mathtt{P} = \mathtt{P}$.
2. Sequential Composition: $\mathtt{T} = \mathtt{P};\mathtt{Q}$. The process T behaves like the process P followed by Q.
3. Conditional Composition: $\mathtt{T} = \mathtt{P}\langle v\rangle : \mathtt{Q}_v$. T behaves like the process P until that terminates. If P aborts, then T aborts. If P stops, then the value v produced by P is acquired, and T then behaves like $\mathtt{Q}_v$.
4. Disabling Composition: $\mathtt{T} = \mathtt{P}\#\mathtt{Q}$. T behaves like the concurrent composition of P and Q until either terminates, then the other is aborted.
5. Synchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v\rangle :; \mathtt{Q}_v = \mathtt{P} : \mathtt{Q};\mathtt{P} :; \mathtt{Q}$. This operator behaves like a while P do Q loop.
6. Asynchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v\rangle :: \mathtt{Q}_v = \mathtt{P} : (\mathtt{Q}, \mathtt{P} :: \mathtt{Q})$. This operator spawns off a instance of Q whenever P terminates; a while P spawn Q loop.

- $\mathtt{P} \xrightarrow{\Omega\mathtt{P}} \mathtt{STOP}$ (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.'
- Concurrent: $(\mathtt{P^0} \mid \ldots \mid \mathtt{P^i} \mid \ldots \mid \mathtt{P^n}) \longrightarrow (\mathtt{P^i} \mid \ldots \mid \mathtt{P^n})$ iff
  $\pi(\Omega\mathtt{P^0}, \ldots, \Omega\mathtt{P^{i-1}})$, where $\pi(A, \ldots, B)$ is the disjunction of all orderings (including conjunction) of the conditions $A, \ldots, B$. The network evolves to STOP if all processes stop and to ABORT if any one aborts.
- Disabling: $(\mathtt{P^0}\#\ldots\#\mathtt{P^i}\#\ldots\#\mathtt{P^n}) \longrightarrow \mathtt{STOP}$ iff $\Omega\mathtt{P^i}$ for an $i \in \{1 \ldots n\}$, in which case for all $j \neq i$ $\mathtt{P^j} \longrightarrow \mathtt{ABORT}$. The network evolves to ABORT if all processes abort.

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