# A Representation for Error Detection and Recovery in Robot Task Plans[1]

D. M. Lyons
Manufacturing Systems Research Dept.
Philips Laboratories
North American Philips Corporation
Briarcliff Manor, NY 10510

R. Vijaykumar
Dept. of Computer Science
Univ. of Southern Maine
Portland, ME 04103

S. T. Venkataraman
Robotics Group
Jet Propulsion Labs.
Pasadena, CA 91109

ABSTRACT

A flexible robotic assembly cell will have to deal with error detection, diagnosis and recovery. Much of the literature in this area to date has concentrated on devising *specific* algorithms for detection and recovery, and has largely ignored the central issues of (1) determining what needs to be represented in order to deal with the problem of detecting and recovering from errors and (2) building an appropriate general representation.

We begin our work by defining the components of the error detection and recovery problem and by defining what kinds of errors can reasonably be detected. We then construct a generally applicable representation for error detection and recovery information that supports these definitions. The representation, which is based on our previous work on task grammars[15], is similar to Donald's[3] notation of an EDR plan. We use $\mathcal{RS}$, a process-based model for representing task plans[10], as the underlying 'vocabulary' to describe our representation concisely.

Our approach yields a solid way to distinguish planned decision making from error monitoring. It also yields a method for determining how to hierarchically structure error monitoring and diagnosis of error. Finally, it suggests a way to monitor the error monitoring system itself.

## 1 Introduction

A flexible robotic assembly cell will typically have to deal with many situations that deviate from the norm: For example, assembly components may arrive late or may break during an operation.

[1]A shortened version of this paper will appear in the 1989 SPIE Symposium on Advances in Intelligent Robotics Systems; Intelligent Control and Adaptive Systems.

Such situations are easily handled by human assembly workers. However, because the set of possible deviations is large, these situations are not easily handled by traditional automation techniques[5]. A number of alternative approaches have been suggested to handle this problem of error detection and recovery in robot task plans[3, 5, 6, 8, 14]. This literature is, by and large, concerned with devising specific methods of detection and recovery. None of these approaches directly address the *central* issues of (1) determining what needs to be represented in order to deal with the problem of detecting and recovering from errors and (2) building an appropriate general representation. This paper will address these central issues.

## 2 Overview and Relevant Literature.

Robot systems act to achieve a goal in two (possibly interleaved) phases: (1) They prepare a *plan* of actions to achieve the goal, based on internal models of the environment in which the activity is to be conducted, and then (2) they execute that plan in the actual environment. In so-called *reactive planning*[1], the plans and models are hard-wired into the system, eliminating the planning phase. In either case, because the internal model might be incomplete or incorrect, the plan execution may not achieve the desired goal. Detecting deviations from the expected in plan execution is called *error detection* or *monitoring.* Finding the cause of the error is called *error diagnosis* (the cause of the error is sometimes called the *fault*). Rectifying the cause of the error is called *error recovery.*

Work in this area of error detection and recovery in robot task plans has concentrated on error diagnosis. Taylor & Taylor[14] introduce a probabilistic model for relating sensor measurements, errors and faults. Lee et al. [7] describe a model-based causal reasoning approach to assembly cell diagnosis, and later Lee & Liu[8] extend this approach to include case-based reasoning for diagnosis. Kumaradjaja et al.[6] describe how a causal model can be used for error monitoring as well as error recovery.

Some work has addressed the question of what needs to be represented. Gini et al.[4] describe an architecture for robust robot control that includes error monitoring, diagnosis and recovery and the relationships between them. However, their domain is somewhat specialized — the automatic augmentation of programmer generated AL (a robot programming language) code with error monitoring clauses — and their architecture is similarly specialized. Cox [2] describes how C++ can be used to represent some important cases of error monitoring, namely errors at initialization, and termination. Other important aspects of the problem, such as diagnosis and recovery and their relationship with monitoring, are not described. Donald[3] proposes to construct motion plans that always succeed or fail recognizably in the presence of geometric uncertainty (which he calls *EDR* plans). Donald pays little attention to how his EDR plans are represented, and instead emphasizes a configuration space approach to EDR plan production. We will apply his EDR concept together with ideas similar to Gini et al.'s[4] robust control architecture to build a generally applicable representation for error detection and recovery information.

An important, and in general missing, part of the current literature on error detection and recovery is the statement of what needs to be represented, i.e. the problem description. We begin our work by defining the components of the error detection and recovery problem and by defining what kinds of errors need to be captured. We then construct a representation for plans that supports these definitions. We argue this is a necessary step *before* working on algorithmic solutions to the components of the error detection and recovery problem.

We begin by defining the error detection and recovery problem domain. We then introduce the representation that we have developed and discuss how it can be applied to the problem. The $\mathcal{RS}$ model is used to implement our representation. The issue of closure of $\mathcal{RS}$ composition operations over the principle definition of our representation is used to show that our approach provides a good method to construct local error monitoring, and to show that $\mathcal{RS}$ process composition operations do not always produce good plan composition operations. We solve this latter problem by defining a number of useful plan composition operations as 'macros' of process composition operations. We then go on to use these macros to demonstrate how various kinds of error diagnosis and recovery can be constructed. In particular, we show how local diagnosis can be build, and how it is related to local error monitoring. Finally, we show how the error monitoring system itself can be monitored by using the notion of monitoring consistency constraints.

## 3 What is Error Detection and Recovery?

In this section, we present a general definition of the error detection and recovery problem in terms of the components of the problem and their inter-relationships.

In its general sense, an error is a deviation of a measurement from an expected value. Clearly, errors have to do with 'measurements' and with 'expectations'. But this definition can lead to ambiguities: As soon as an 'error' test is embedded in a plan, it becomes part of the 'expected' measurements in that plan, and thus can no longer be classified as an 'error' test!

The intuitive notion of an 'error test' remains clear despite this ambiguity: the 'extra' error testing code is logically separate and distinct from the plan which it monitors. The meanings of 'expected' and 'measured' in the definition of an error *have* to be made with respect to the plan being monitored. This important separation of error testing from planned decision making is a clear part of the problem definition, but it has not been achieved in the field up to now (Donald's work[3] comes closest, but he concentrates on a planning algorithm without too much concern for what his EDR notion implies for plan representation). Our plan representation distinguishes planned decisions from error monitoring of the plan.

It is common in the literature to distinguish an error from a *fault* or *cause* as follows: an error is a measured deviation that is *caused* by a fault. This is similar to the relationship between a symptom and a disease. Thus a fault may not be directly observable, but its effects, errors, can be observed. Whereas an error is a measurement, a fault is a state of the environment, e.g., an unpredicated state produced by an action. A fault may remain undetected for some time after it occurs (i.e., it may not immediately cause an error). In addition, a fault that is not rectified can cause other faults and thus start a 'snowballing' of errors. Relating errors to faults is generally called *diagnosis*.

There is, in general, a chain of events that are related causally that lead from the fault to the observed error. As to whether an error can be diagnosed to the fault that caused it, or merely an event in this chain is a function of the amount of knowledge present in the diagnosis system. A fault can be classified as *recoverable* or *unrecoverable* depending on whether or not a recovery procedure can be carried out that enables the original goal to be achieved. Recovery can be divided into two main classes: (1) restoring the environment to a state in which the plan can be resumed or restarted (*plan patching*), and (2) producing a new plan to take the environment from the fault to the original goal (*replanning*).

It is necessary to understand for what kinds of errors and faults our problem description holds. The crucial characteristic is that in order for our definition of error detection to make any sense, there must be a correct or 'expected' behavior of the system. This supports the reasonable demand that we should be able to monitor and recover from errors caused by failure of the robot hardware, by failure of external machines, or due to uncertainty in the events and geometry of the task. However, it will not support monitoring and recovering from errors in the design of the workcell, or programmer errors in the cell software, since no correct 'expected' behavior can be defined in these cases.

Monitoring errors requires depending on some components of the system to reliably produce the input for the error measurements. It is not possible, in general, to detect errors *of* the error monitoring system, since these are akin to errors in the design of the workcell. However, it should be possible to check the data on which the error monitoring system makes its decisions. This can be done by setting forth a set of *consistency assumptions* about the environment and monitoring that these assumptions are not violated in any data reported by the sensory system. For example, a reasonable visual consistency rule is that objects cannot change their identity spontaneously. Should this happen to an object during an operation, then we know our sensory system has failed, and we should not allow error monitoring to operate on data from this source. Our plan representation supports this form of inconsistency detection.

# 4 Task Plan Representation

This section introduces the representation that we have developed to deal with the error detection and recovery problem. First, we present the definition of a task plan from Vijaykumar et al.[15].

## 4.1 Components of a Task Plan

Informally, a task plan can be defined as a *domain of interaction between the robot and the environment.* Associated with it are a purpose, the *goal*, a method, the *control strategy*, and a specification of when the method remains applicable, the *task invariant.* The task control strategy is a description of the relationship between sensing and action in the plan; it represents one possible way to achieve the goal of the plan. The control strategy captures planned decision-making. However, the environment may not support the use of the control strategy to achieve the goal. Donald[3] gives an example for peg in hole insertion: the control strategy may contain sensing and decision-making to move the peg compliantly to accommodate contact with the sides of the hole during insertion. However, if there is uncertainty about the size of the hole, then in the case that the hole is too small, the control strategy will just not work. The purpose of the task invariant condition is to continually monitor those conditions that ensure that the control strategy will succeed.

**Definition 1** *A task plan $T$ is specified by the following*

- *$G$ represents the goal the task is to achieve as a test on the state of the environment.*
- *$C$ represents the control strategy of the task: a relationship of sensory input and motor output.*
- *$I$ represents the task invariant as a test on the state of the environment that determines if $C$ will still result in the achievement of the goal $G$.*
- *These components should obey the rule that if $I$ remains true while $C$ is being carried out, this implies $G$ will eventually be achieved[2]:*

$$\boxed{\text{I holds while doing(C) IMPLIES eventually G}} \tag{1}$$

This definition clearly fulfills the EDR definition[3]: $T$ will either succeed ($G$ achieved) or fail recognizably ($I$ violated).

### 4.2 Characteristics of Error Monitoring with Invariants

Planned Decisions vs. Error Monitoring. This task plan representation partitions planned decision making from error monitoring: The control strategy is clearly the pre-planned decision making component. In some appropriately simplified (or 'ideal') world, the control strategy is *guaranteed* to succeed in achieving the goal; in the 'real' world, it is *predicted* that the control strategy will achieve the goal. In the peg in hole example, the simplified world would be a world in which the uncertainty about the size of the hole would never allow a hole too small for the peg. There is an essential property of the environment that guarantees the accomplishment of the goal in the simplified world, and the extrapolation to the real world can only work if the assumption that the property remains true holds in the real world. This assumption about the environment on which the prediction is based is captured in the invariant. Thus the invariant for the peg in hole example is that the peg fit in the hole. The validity of the invariant clearly determines if there is divergence between the real and simplified worlds: it's purpose is clearly error monitoring. We have achieved our objective of separating planned decision making from error monitoring.

Efficient Error Monitoring. Error Monitoring using invariants is a far more efficient and directed approach than basing error monitoring on a set of likely errors (e.g., Hardy et al.[5], among many others). This is because the invariant monitors the assumptions that are expected to hold in order for the control strategy to succeed in achieving the goal, and doesn't prematurely proscribe the errors that might cause the strategy to fail. In general, the set of errors that can cause a plan to

[2]By implies, we mean causal implication due to the set of causal laws that hold in the environment.

fail is open-ended. The invariant approach is not to monitor this open-ended set, but instead, the more well-defined domain of situations under which the control strategy will achieve the goal.

Simplified Error Diagnosis. Error monitoring using invariants facilitates the process of error diagnosis. This is due to the fact that when the invariant fails, the deduction that the system will not achieve its goal is not solely based on the manifest behavior of the system. It is based, in addition, on the essential premise on which the accomplishment of the goal is predicated. While the invariant may not directly identify the fault, the diagnosis system only needs to deduce the fault from the failure of the invariant. This is similar to the use of precondition reasoning in diagnosis[6, 4].

### 4.3 Implementing The Task Plan Definition

To evaluate the usefulness of Definition 1, we need to map it onto a concrete framework that will support computer implementation and facilitate mathematical analysis. In[10, 9] we describe an approach, the $\mathcal{RS}$ model, to representing sensory-based task plans as networks of parallel, communicating processes. That approach offers a representation for sensory-based plans, an efficient approach to executing plans (based on distributed processing), and an underlying mathematical framework for analyzing plans. The 'vocabulary' of $\mathcal{RS}$ is a set of concurrent processes and a set of operators that can be used to compose the processes into networks. This vocabulary allows us to give concrete meaning to the components of a task plan that respects Definition 1.

We have investigated using two other common plan representations for representing a plan that respects Definition 1: the Hypergraph model[**?**] and the Petri-Net model[13]. The AND-OR links of the hypergraph model have to be extended in an ad-hoc fashion to represent the sensing (e.g. measuring the goal and invariant conditions as well as sensing within the control strategy) and timing (constraint (1)) in Definition 1. The Petri Net model will represent the sensing and timing constraints, but the style is as a *static* network that emphasizes flow of control. Definition 1 suggests a *dynamic* formulation of task plans; one in which the components of the plan and their interconnections vary over time depending on what conditions hold in the environment. It also emphasizes flow of values or information between, for example, the sensing and motor components of a control strategy, or between diagnostic and recovery components. The $\mathcal{RS}$ model subsumes the Petri-net model, and also can represent flow of information as message passing across communication ports[3] in a set of concurrent processes. The $\mathcal{RS}$ process composition operators allow the construction of dynamic networks of processes, networks whose connections and components can change over time. For these reasons, we chose $\mathcal{RS}$ as the implementation for plans that respect Definition 1.

The 'vocabulary' of $\mathcal{RS}$ is a set of concurrent processes and a set of operators that can be used to compose the processes in various ways (Appendix A is an overview of these operators; they will be introduced as needed in the remainder of the paper). The three components of Definition 1 can each be captured as a process: `C` can be considered a servoprocess implementing the control strategy $C$, `I` can be considered a process monitoring the invariant condition $I$ and terminating

[3]Including fan-in and fan-out at ports.

the task if it becomes false, and $\texttt{G}$ can be considered as a process monitoring the goal condition and terminating the task if it becomes true. The $\mathcal{RS}$ composition operator '#' composes a number of processes together into a network in which they are all concurrent and in which if any one terminates, then the network is terminated (Appendix A). We can now ground our definition of $\texttt{T}$ as an $\mathcal{RS}$ plan schema as follows:

$$\texttt{T} = (\texttt{I}\#\texttt{G}\#\texttt{C}). \tag{2}$$

$\texttt{T}$ is a network of process consisting of a non-terminating servoprocess $\texttt{C}$ and two monitor processes, one that terminates $\texttt{T}$ on invariant failure, and one that terminates $\texttt{T}$ when the goal is achieved. Constraint (1) in Definition 1 guarantees that one of these two events will eventually occur for any plan $\texttt{T}$.

Not all plans terminate when their goal is achieved. There is a useful class of plans whose purpose is to *maintain* some condition in the environment for as long as the plan is active. We will call these *maintenance* plans; examples include maintaining a gripping force on an object in order to keep it within the gripper, or tracking an object with a camera. We can use the plan schema (2) to represent maintenance plans as follows: Let $\bar{\texttt{G}}$ denote a process that terminates only when the goal condition $G$ is *not* satisfied by the environment. We can use this process to continually trigger a plan to achieve $G$:

$$\texttt{T} = \bar{\texttt{G}} :; (\texttt{I}\#\texttt{G}\#\texttt{C}). \tag{3}$$

where ':;' is the $\mathcal{RS}$ composition operator that activates its first argument, and when that terminates, activates the second, and when that terminates, reactivates the first, etc. (Appendix A). $\bar{\texttt{G}}$ ensures that the plan to achieve $G$ is triggered whenever $G$ does not hold. The plan only terminates if $G$ is achieved or if there is an error. We shall deal with the error case presently. If the goal is achieved, then the plan is suspended and the goal trigger $\bar{\texttt{G}}$ is reactivated.

Note that this formulation actually *restores* the the goal rather than maintains it — countering, rather than preventing, disturbances. However, the goal trigger could be tied to an signal that the goal is *about* to change, rather than to the event of the goal changing. In that case, the control strategy would be activated in time to prevent the goal changing (e.g. applying a force to an object to keep it stationary in the face of an oncoming collision).

<u>Example.</u> As an example, consider a plan $\texttt{Tp}$ for a guarded move. Call the robot end-effector $E$ and the desired contact surface $D$. The goal of the task plan is

$$\texttt{Gp} : E\ \textit{against}\ D.$$

where *against* represents the geometric relationship of the end-effector in contact with the desired surface. The control strategy for this task plan is a compliant move expressed, for example, as

$$\texttt{Cp} : u = q(k\Delta(r) - f).$$

The equation expresses the relationship between control signal $u$ that drives the end effector towards $D$, as the difference between the sensed force $f$ and the *compliance* force $k\Delta(r)$ caused by the difference in end-effector position $\Delta(r)$. $q$ and $k$ are constant terms. The control strategy only works if, for example, there exists a collision free line from the end effector's present location, $r$, to the contact surface $D$.

$$\texttt{Ip} : \neg\exists\ \textit{an object between } r \textit{ and } D.$$

The task plan

$$\texttt{Tp} = (\texttt{Ip}\#\texttt{Gp}\#\texttt{Cp}).$$

executes the strategy `Cp` but terminates if either the invariant fails or if the goal is reached. Definition 1 tells us that if `Cp` is executing, and `Ip` does not terminate, then eventually `Gp` will terminate the task successfully. We can extend this plan to *maintain* the end-effector in contact with the surface by introducing a goal trigger process that is simply

$$\bar{\texttt{G}}\texttt{p} : \neg(E \ against \ D).$$

and extending `Tp` to be a maintenance plan (3) as follows:

$$\texttt{Tp} = \bar{\texttt{G}}\texttt{p} :; (\texttt{Ip}\#\texttt{Gp}\#\texttt{Cp}).$$

# 5 Plan Hierarchy

It is very convenient and useful to be able to describe a plan as a composition of other more primitive plans. Indeed, it would be very difficult to build complex plans unless this sort of abstraction was possible. $\mathcal{RS}$ supports this hierarchical abstraction; a network of processes can be included as a single process within some other ('higher-level') network of processes. This gives us a mechanism to build hierarchical plans, but this on its own is not sufficient. We also need to show that hierarchical composition is *closed* with respect to Definition 1; that is, if we build a composite plan from more primitive plans, each of which respects Definition 1, then the composite plan also respects Definition 1. We start off by looking at closure over the $\mathcal{RS}$ process composition operators as described in Appendix A. This will give us a useful concept of locality in error monitoring and recovery. However, the process composition operators do not always map well onto useful plan compositions. Thus, in this and the following section, we develop some useful plan composition operators, and define them as process composition 'macros.'

For example, consider the network `Grasp` described as follows in terms of more primitive processes:

$$\texttt{Grasp} = (\texttt{Reach}, \texttt{Preshape}) \ ; \ \texttt{Grip} \ ; \ (\texttt{Move}\#\texttt{Release}).$$

This consists of the processes `Reach` and `Preshape` run in parallel (this is what the network composition operator ',' does, see Appendix A) followed by the process `Grip` when these two terminate (this is what the sequential composition operator ';' does, see Appendix A). When `Grip` terminates, the network of `Move` and `Release` is created and if either terminates then both terminate (disabling composition, see Appendix A).

Informally[4], let `Reach` and `Preshape` concurrently control the reaching of a robot arm to a target object, and the preshaping of the hand to suit the object size and shape, respectively. When they complete, let the `Grip` process close the fingers to stably grip the object. Finally, let `Move` manipulate the object in some prescribed way while `Release` determines when it is appropriate to

[4]We describe this network in the context of an implementation in Lyons&Mehta[12].

In each of the following, $\texttt{Ti} = (\ \texttt{Ii} \# \texttt{Gi} \# \texttt{Ci}\ )$ , $\wedge$ is the logical AND operator, $\vee$ is OR, and *then* is the temporal precedence operator. For technical convenience, we use the same name for the condition and for the process that tests for this condition.

- **Network Composition:** $\texttt{T} = (\texttt{T1}, \texttt{T2}\ldots)^c$
  Composite Invariant: $\texttt{I1} \wedge \texttt{I2} \wedge \ldots$
  Composite Goal: $\texttt{G1} \wedge \texttt{G2} \wedge \ldots$

- **Sequential (and Enabling) Composition:** $\texttt{T} = \texttt{T1}; \texttt{T2}$
  Composite Invariant: $\texttt{I1}$ *then* $\texttt{I2}$
  Composite Goal: $\texttt{G1}$ *then* $\texttt{G2}$

- **Disabling Composition:** $\texttt{T} = \texttt{T1}\#\texttt{T2}$
  Composite Invariant: $\texttt{I1} \wedge \texttt{I2}$
  Composite Goal: $\texttt{G1} \vee \texttt{G2}$

Note that the conditions in network composition can become true in *arbitrary* order.

**Table 1: Results of Plan Composition on Invariant and Goal Conditions.**

carry out the action to release the object and terminate the network. Now, to investigate closure of Definition 1 under composition we ask: if all the processes are task plans in the form of Definition 1, then what does the goal and invariant of the entire network look like?

It is possible to derive the composite invariant and goal conditions of a task plan when the control strategy of the plan is composed of other plans. In Appendix B, we present a formal derivation of composite goal, invariant and control strategy in terms of the goal, invariant and control strategies of the component plans. If we simplify these results by looking only at the interactions between component goal or between component invariant processes, then we get the relationships between composition operation and composite goal or invariant process shown in Table 1. This simplification is useful when we are just interested in the overall goal or invariant condition that a composite plan tests. In the `Grasp` example, this corresponds to determining under what conditions the entire network will fail (the invariant condition) or will succeed (the goal condition). While this is useful for plan analysis, we will also show that it is useful in constructing localized error monitoring. Given this table, we can define a straightforward algorithm to yield the invariant and goal conditions of a network from the components of that network.

Consider first the invariant condition: Let $Itrans(op, \texttt{T1}, \texttt{T2}, \ldots)$ be a function that takes a composition operation *op* and a set of component processes $\texttt{T1}, \texttt{T2}, \ldots$ and returns the logical combination of the invariants of the component processes caused by *op*. That is, it is a function that 'looks up' Table 1 for us, e.g., $Itrans(`\#', \texttt{T1}, \texttt{T2}) = \texttt{I1} \wedge \texttt{I2}$. We can now define a function

**Figure 1: The Goal and Invariant hierarchies induced by a hierarchical task plan for grasping, `Grasp` .**

$Icond(\texttt{T})$ that returns the invariant condition for *any* plan `T` as follows:

$$Icond(\texttt{T}) = \begin{cases} \texttt{I} & \text{If } \texttt{T} \text{ is a primitive plan.} \\ Itrans(op, Icond(\texttt{T1}), Icond(\texttt{T2}), \ldots) & \text{If } \texttt{T} \text{ has components } \texttt{T1,T2,...} \\ & \text{composed using } op. \end{cases}$$

It is clearly possible to define the analogous functions $Gtrans()$ and $Gcond()$ to pull out the goal condition for a composite task plan. Thus we have shown that Definition 1 is closed over composition. In addition, we also now have a way to describe error detection in a hierarchical fashion. This will form the basis of our approach to *localizing* error detection and recovery.

Figure 1 illustrates the way in which the `Grasp` plan, which is composed hierarchically, induces a hierarchical structuring on the goal and invariant monitoring. The invariant of the network, $Icond(\texttt{Grasp})$, is derived as follows. The `Grasp` network consists of three processes composed sequentially. All the composition operators are associative[11], our graphical convention here will be to work in from the left towards the right. Thus, in Figure 1 the topmost level of `Grasp` is a sequential composition node with two arguments. The first argument (the left-most part of `Grasp`) is itself a network composition of `Preshape` and `Reach` . The second argument is a sequential composition of `Grip` and a disabling composition. The disabling composition has arguments `Move` and `Release` . The 'effect' of applying $Itrans()$ and $Icond()$ is to relabel the internal nodes of this hierarchy with logical operations corresponding to the composition operations, and relabel the leaves with the name of the invariant process of the primitive plans. Note that Figure 1 shows the translation between adjacent levels in a plan hierarchy. The root nodes in the figure represent the upper-level plan, the leaves are the lower-level plans. This is distinct from the tree structure that composition operations will impose on a network at any level, i.e., the internal nodes in the figure.

Local Error Monitoring. It is important to understand the hierarchical implications of Definition 1. If task plan $\mathtt{T} = (\mathtt{I}\#\mathtt{G}\#\mathtt{C})$ and if `C` is hierarchical (e.g., `C=(T1#T2)` ), then some error monitoring is already being done *in* `C`, namely $Icond(\mathtt{C})$ (e.g., $Icond((\mathtt{T1}\#\mathtt{T2}))$). This is very advantageous: Not every condition needs to be explicitly measured everywhere, the invariant process `I` needs to measure *only* those condition relevant to its level of the hierarchy, and $Icond(\mathtt{C})$ measures conditions below this level. For example, in a plan that consists of grasping an object and conveying it to a parts-mating operation, the plan should test for correct orientation of the grasped object (since this is important to being able to start the parts-mating), but not for whether the object was acquired or not (this should be tested for by the grasping subplan).

We can minimize the duplication in error monitoring at different levels as follows: When we are constructing the invariant process `I` for some plan `T`, we consider the overlap between $Icond(\mathtt{C})$, the monitoring done (hierarchically) within the control strategy, and $I$ the invariant at that level from Definition 1. The invariant process `I` need only monitor the *difference* between these two conditions, that is, the conditions in $I$ that are not tested already in $Icond(\mathtt{C})$. We write this as

$$\mathtt{I} : \ I - Icond(\mathtt{C}).$$

The invariant for `T`, the sum total of all monitoring in `T`, is now $Icond(\mathtt{C}) \wedge Icond(\mathtt{I})$, where `I` is now the *hierarchically complete invariant* for `T`.

Chaining Plans. The $\mathcal{RS}$ composition operators do not always produce good plan compositions. For example, a sequential composition of two plans means that the two plans are *always* executed sequentially, no matter what invariant failures or goal successes occur. However, we can use the $\mathcal{RS}$ operators as building blocks with which to construct useful plan composition operations.

It is useful to be able to specify a chain of plans that are conditionally sequenced. That is, the next plan in the chain is only carried out if the previous plan met its goal. Our definition of the success of a plan is that its goal process `G` terminates. Since we have a concrete definition of success, we can define a goal chained plan as follows:

$$Gchain(\mathtt{T1}, \mathtt{T2}) = (\mathtt{T1}, \mathtt{I1}\#(\mathtt{G1} : \mathtt{T2})).$$

The enabling composition operator ':' is a special kind of sequential composition in which the first argument can pass on some parameters its successor process (we will need the parameter facility presently, but it is not used above). The successful chaining of `T1` and `T2` is defined as the network of `T1` concurrent with the goal monitor process of `T1` enabling the execution of `T2`. That is, when `T1=(I1#G1#C1)` has successfully achieved its goal, `G1` terminates and triggers `T2`. However, if `I1` terminates first, then it prevents the triggering of `T2`. The `G1` and `I1` processes are the *same* as the goal and invariant processes inside `T1`. A clever implementation would unify these processes instead of creating two copies of each. (That is, however, an implementation issue.)

## 6 Recovering from Errors

So far, only error monitoring has been covered. We now show how diagnosis and recovery can be handled via the invariant representation. The connection between error monitoring and recovery can be made by chaining the recovery procedure with the invariant monitoring process. If `R1` is a recovery procedure for `T1`, we can connect them as follows (by analogy with $Gchain()$ above):

$$Ichain(\mathtt{T1}, \mathtt{R1}) = (\mathtt{T1}, \mathtt{G1}\#(\mathtt{I1} : \mathtt{R1})).$$

In this case, `R1` is only ever enabled when `I1` terminates. However, if `G1` should terminate first, then the error recovery link is disabled. We can now pay more attention to what `R1` actually does.

Using this framework, we can now build the kinds of error diagnosis and recovery that we described in the problem overview section (Section 3).

1. *Canned recovery:* It is sometimes sufficient to have 'canned' recovery procedures `RT` associated with each task plan `T`. This is simply $Ichain(\mathtt{T}, \mathtt{RT})$. If we wish to retry the action after recovery, we can write this recursively as $\mathtt{T} = Ichain(\mathtt{T1}, \mathtt{RT}; \mathtt{T})$.

2. *Diagnosed Recovery:* In general, it is preferable to have some diagnosis of the error before recovery commences
$$\mathtt{T} = Ichain(\mathtt{T1}, \mathtt{D}\langle f \rangle : \mathtt{R}_f).$$
In this scenario, if the plan fails, a special diagnosis procedure `D` is called. Diagnosis reduces the error to a fault, $f$. The distinguishing characteristic of a fault is whether it is *recoverable* or not. We can associate a recovery procedure $\mathtt{R}_f$ with each recoverable fault $f$. We can then use the output of `D` to choose which $\mathtt{R}_f$ to execute to recover. We have two choices once recovery has finished:

   (a) *Plan Patching:* We can restart or resume the plan at an appropriate point, for example, just before the failed action:
   $$\mathtt{T} = Ichain(\mathtt{T1}, \mathtt{D}\langle f \rangle : \mathtt{R}_f; \mathtt{T})$$
   In the above plan, `T1` is recursively restarted after recovery has finished. Nesting of $Ichain()$ and $Gchain()$ can be used for even greater flexibility in resuming plans.

   (b) *Replanning:* We can jump back to the planner itself. The planner itself can simply be characterized as a process `Planner` that generates the new plan, *plan*, that is then executed:
   $$\mathtt{T} = Ichain(\mathtt{T1}, \mathtt{D}\langle f \rangle : \mathtt{R}_f \ ; \ (\mathtt{Planner}\langle plan \rangle : \mathtt{Execute}_{plan})).$$
   We could also choose to omit any recovery (or even diagnosis) step and jump directly back into the planner whenever an error occurs.

<u>Graceful Recovery.</u> We have to consider what happens if the fault is unrecoverable. The hierarchical recovery approach of Lee et al.[7] is very useful: Should an error not be recoverable at a particular

level in the plan, then the error recovery is 'shipped' to the next higher level. This gives us a way to reduce the complexity of diagnosis at each level to just the conditions monitored by the invariant at that level according to our hierarchical invariant definition (Section 5).

Let us consider the world to be characterized at any point in time by a state vector $W$. The conditions monitored by the goal and invariant processes are conditions on elements of $W$. $TV()$ be a function that projects an invariant process onto the elements of $W$ that the invariant condition tests. For example, if `I` just monitored a constraint on a force sensor $f$ and a position sensor $p$, then $TV(\mathtt{I}) = (f, p)$. We will call the results of applying $TV()$, the *task variables* of the invariant.

The task variables monitored by the invariant at each level are the *only* variables that should be diagnosed at each level. We only need to extend our diagnosis module to accept a set of task variables to work on as a parameter — the diagnosis process is now $\mathtt{D}_{TV(\mathtt{I})}$ which has initial parameters $TV(\mathtt{I})$, the task variables of the invariant process `I`. At each level, the diagnosis and recovery representation is as follows:

$$\mathtt{T} = Ichain(\mathtt{T1}, \mathtt{D}_{TV(\mathtt{I1})}\langle f\rangle : \mathtt{R}_f).$$

And if the error is not be recoverable at this level, then we allow the task plan `T` to terminate unsuccessfully, and let the diagnosis be continued by the higher level (in the hierarchy) plan of which `T` is a component. This is what we call *graceful recovery*. It fits in well with the concept of hierarchical monitoring introduced in section 5. In that section, we described how to derive what to measure in the invariant at a level in the hierarchy. Graceful recovery relates what is being measured in the invariant at a level in the hierarchy to what diagnostic reasoning should be done at that level.

## 7 Error Monitoring by Consistency

It is difficult to do error monitoring on the error monitoring system itself. One solution is to have multiple redundant monitoring systems and have them vote on the results. Another approach is attempt to make the information coming into the error monitoring system more reliable, that is, to monitor the sensory system.

As an example, consider the *misrecognition problem*: A vision system identifies an object based on the data it has seen so far. It is possible that an object might be misidentified because the parts of the objects seen by the sensory system up to that point do not uniquely identify the object or because of noise in the data. In such a case, the error monitoring system is handicapped, because the source of error measurements, the sensory system, is itself at fault. The error monitoring being carried out is simply no longer appropriate, but there is no way the error monitoring system can tell this — it has to rely on the veracity of the sensor data. We have found in our experience of coupling a sensory system to a task planning and control system that such a problem is not unusual.

We can catch this type of error by having processes that monitor the *consistency* of the data coming back from the sensory system. The environment monitored by the sensory system operates according to a set of causal laws. The data produced by the sensors should, therefore, be consistent with these laws. If an inconsistency is detected, then clearly, the sensory system has been in error. The solution we have adopted in the misrecognition problem relies on consistency of object identity: An object in the robot environment should not spontaneously change its identity. A block should remain a block, a bolt should remain a bolt. Should an object appear to violate this consistency constraint, then we know the sensory system is suffering from an 'illusion' and should be ignored.

Our approach is to associate an invariant condition with each object that has been recognized. If the object is an instance of object model $o$, then we represent it as a process `Obj` with parameter $o$. The process $\texttt{Obj}_o$ reports the sensory data that has been gathered about the object; by analogy with `C` in definition 1, this process will never terminate. We represent the invariant as a process `Iobj` with parameter $o$. $\texttt{Iobj}_o$ monitors the consistency of the data reported by $\texttt{Obj}_o$ ; by analogy with `I` of definition 1, it terminates when the consistency constraint is violated.

$$\texttt{Robust.Object}_o = (\texttt{Obj}_o \# \texttt{Iobj}_o).$$

$\texttt{Obj}_o$ is the process that channels the sensory systems data on the object into the task. The invariant process, $\texttt{Iobj}_o$, continually (or for efficiency, periodically) asks the sensory system to re-identify the object that it is 'guarding.' Should the consistency constraint be violated, i.e., should the invariant process find that the sensory system reports that the identity of the object represented by $\texttt{Obj}_o$ has changed from $o$ to some other model, then the invariant terminates, causing the object process to be killed.

As described in the error recovery section, the invariant can be used to tie into mechanisms for recovery. In this case, it is appropriate to reinvoke recognition to again search for an instance of the desired object model $o$. By analogy with the definition of $Ichain()$ for task plans, we can redefine `Robust.Object` as

$$\texttt{Robust.Object}_o = (\texttt{Obj}_o \# \texttt{Iobj}_o,\ \texttt{Iobj}_o\ ;\ (\texttt{Identify.Object}\langle p \rangle : \texttt{Robust.Object}_p)).$$

If we make the assumption that when `Identify.Object` cannot find an appropriate object model instance it waits until one arrives, then $\texttt{Robust.Object}_o$ has the interesting property that it robustly represents an instance of object model $o$.

# 8 Conclusion

Our approach to the problem of error detection and recovery in robot task plans has been to (1) determine what needs to be represented in order to deal with the problem of detecting and recovering from errors and (2) build an appropriate general representation. Much of the literature in this area is, on the other hand, largely concerned with specific methods for diagnosis and recovery.

In section 3 we present a definition of the problem domain. Our fundamental representation of a task plan is described in Definition 1. This definition is similar to that of Gini et al.[4] in that it consists of a 'monitor' process that monitors the task in progress. One difference is that our monitor, the task invariant, is concurrent and there is one such monitor per plan. The relationship between the task invariant and the control strategy is given by constraint 1 of Definition 1. This constraint is very similar in effect to Donald's[3] notion of an EDR plan.

We use the $\mathcal{RS}$ model to give a concrete language in which to describe our representation. We chose this model because it represents the timing and sensing details of Definition 1 better that other approaches, because it translates to an efficient distributed implementation[12], and because it facilitates analysis and evaluation of our representation[11].

Donald considers his EDR plans as monolithic entities. By 'breaking' an EDR plan into the goal, strategy and invariant components, and expressing these in $\mathcal{RS}$, we can make some concrete statements about hierarchical error monitoring and about diagnosis, and recovery. In addition, we can extend the bounds of error monitoring using our approach, and apply consistency monitoring to the inputs of the error monitoring system.

In addition, our approach now suggests a novel way to look at the EDR plan generation problem: That is, given a goal and control strategy, generate the invariant such that the plan becomes EDR. We have not yet looked at this problem in any depth, but because it starts off with more information (the strategy and goal are given) than Donald's approach, we feel it may be fruitful.

## References


[1] D. Chapman and P. Agre. Abstract reasoning as emergent from concrete activity. In *Workshop on Planning& Reasoning about Action*, pages 251–266, Timberline OR, 1986.

[2] I. Cox. C++ language support for guaranteed initialization, safe termination and error recovery in robotics. In *IEEE Int. Conf. Robotics & Automation*, pages 641–643, 1988.

[3] B. Donald. A geometric approach to error detection and recovery for robot motion planning with uncertainty. *Artificial Intelligence*, 37(1–3):223–271, Dec. 1988.

[4] M. Gini, R. Doshi, M. Gluch, R. Smith, and I. Zualkernan. The role of knowledge in the architecture of robust robot control. In *IEEE Int. Conf. Robotics & Automation*, pages 561–567, 1985.

[5] D.P. Hardy, N.W. Barnes and M.H. Lee. Automatic diagnosis of task faults in flexible manufacturing systems. *Robotica*, 7(1):25–35, Jan. 1989.

[6] R. Kumaradjaja and F. DiCesare. A causal reasoning approach for planning error recovery in automated manufacturing systems. In *SPIE Symp. on Advances in Intelligent Robotic Systems; Expert Robots for Industrial Use (Vol. 1008)*, pages 144–153, 1988.

[7] C.N. Lee, M.Y. Chiu, P. Liu, and S.J. Clark. Model-based hierarchical diagnosis of robotics assembly cell. In *SPIE Symp. on Advances in Intelligent Robotic Systems; Intelligent Robots and Computer Vision (Vol. 848)*, pages 182–189, 1987.

[8] C.N. Lee and P. Liu. Case-based reasoning for robotic assembly cell diagnosis. In *SPIE Symp. on Advances in Intelligent Robotic Systems; Expert Robots for Industrial Use (Vol. 1008)*, pages 241–246, 1988.

[9] 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.

[10] 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.

[11] D.M. Lyons and I. Mandhyan. Fundamentals of $\mathcal{RS}$ - part II: Process composition. Technical Report TR-89-033, Philips Laboratories, 345 Scarborough Rd., Briarcliff Manor, NY 10510, Jun. 1989.

[12] 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.

[13] J.L. Peterson. *Petri-Net Theory and the Modelling of Systems.* Prentice-Hall, 1981.

[14] G.E. Taylor and P.M. Taylor. Dynamic error probability vectors: A framework for sensory decision making. In *IEEE Int. Conf. Robotics & Automation*, pages 1096–1100, 1988.

[15] R. Vijaykumar, S. Venkataraman, G. Dakin, and D. Lyons. A task-grammar approach to the structure and analysis of robot programs. In *IEEE Workshop on Languages for Automation*, pages 39–42, Vienna, Austria, Aug. 24-27th 1987.

# A $\mathcal{RS}$ Process Composition Operators

A schema is a description of a process or a network of processes. A process in $\mathcal{RS}$ is a parameterized instance of a schema, e.g. the schema $\mathtt{A}$ instantiated with initial parameters $x1, x2, \ldots$ is written $\mathtt{A}_{x1,x2,\ldots}$. Process networks are built up by combining individual processes with process composition operators. The operators in $\mathcal{RS}$ were designed with sensory-based robot plans as target. For examples of how these can be used to build sensory-responsive plans, see [10, 9].

- **Network** or **Assemblage Composition** combines a network of processes connected together via communication ports into a single process:

  $$\mathtt{T} = (\mathtt{A}_i, \mathtt{B}_j, \ldots)^c$$

  The map $c$ specifies the communication paths between the concurrent processes $\mathtt{A}_i, \mathtt{B}_j, \ldots$. The network terminates when all its processes terminate.

- **Sequential Composition** combines two processes 'back to back.'

  $$\mathtt{T} = \mathtt{A}_i ; \mathtt{B}_j$$

- **Enabling** or **Conditional Composition** combines two processes in such a way that the first can 'enable' the second.

  $$\mathtt{T} = \mathtt{A}_i \langle j \rangle : \mathtt{B}_j$$

  When $\mathtt{A}$ terminates the parameter $j$ is used as an initial parameter to $\mathtt{B}$.

- **Disabling** or **Transcondition Composition** combines two processes together as a network in such a way that if either one terminates, the network terminates.

  $$\mathtt{T} = \mathtt{A}_i \# \mathtt{B}_j$$

- **Recurring Conditional Composition** is a *recursive* composition defined as

  $$\mathtt{T} = \mathtt{A} :; \mathtt{B} = \mathtt{A} : \mathtt{B} ; \mathtt{A} :; \mathtt{B}$$

  $\mathtt{A}$ continually 'triggers' copies of the process $\mathtt{B}$. This is the 'synchronous' form of the composition. We can also define an 'asynchronous' form, in which $\mathtt{T}$ does not wait for $\mathtt{B}$ to terminate before being able to trigger again.

## B Closure of Definition 1 with respect to Process Composition

The closure problem can be formally put as follows. Given $\mathtt{T1} = (\mathtt{I1}\#\mathtt{G1}\#\mathtt{C1})$ and $\mathtt{T2} = (\mathtt{I2}\#\mathtt{G2}\#\mathtt{C2})$ can

1. $\mathtt{T} = (\mathtt{T1}, \mathtt{T2})$,
2. $\mathtt{T} = (\mathtt{T1}\#\mathtt{T2})$ and
3. $\mathtt{T} = (\mathtt{T1} : \mathtt{T2})$

each be rearranged to the form $\mathtt{T} = (\mathtt{I}\#\mathtt{G}\#\mathtt{C})$? Considering these three operators is sufficient because the other operators can all be written in terms of these three. We now present the proofs for each of these cases.

Closure over Concurrent Composition. From [11] we get that $(\mathtt{A}\#\mathtt{B}\#\mathtt{C}) = (\mathtt{A}\#\mathtt{B}, \mathtt{B}\#\mathtt{C})$ (let us call this the #-collection rule), and that network composition is commutative and associative. Thus, we have

$$\begin{array}{lll}
(\mathtt{T1},\mathtt{T2}) & = ((\mathtt{I1}\#\mathtt{G1}\#\mathtt{C1}), (\mathtt{I2}\#\mathtt{G2}\#\mathtt{C2})) & \\
 & = ((\mathtt{I1}\#\mathtt{G1}, \mathtt{I1}\#\mathtt{C1}), (\mathtt{I2}\#\mathtt{G2}, \mathtt{I2}\#\mathtt{C2})) & \text{(by \#-collection)} \\
 & = (\mathtt{I1}\#\mathtt{G1}, \mathtt{I1}\#\mathtt{C1}, \mathtt{I2}\#\mathtt{G2}, \mathtt{I2}\#\mathtt{C2}) & \\
 & = ((\mathtt{I1}\#\mathtt{G1}, \mathtt{I2}\#\mathtt{G2}), (\mathtt{I1}\#\mathtt{C1}, \mathtt{I2}\#\mathtt{C2})) & \text{(by rearranging terms)} \\
\multicolumn{3}{l}{\text{But this is of the form:}} \\
\mathtt{T} & = (\mathtt{I}\#\mathtt{G}, \mathtt{I}\#\mathtt{C}) & \\
 & = (\mathtt{I}\#\mathtt{G}\#\mathtt{C}) & \text{(by \#-collection)}
\end{array}$$

Closure over Disabling Composition. The disabling composition operator is both associative and commutative [11].

$$\begin{array}{lll}
(\mathtt{T1}\#\mathtt{T2}) & = ((\mathtt{I1}\#\mathtt{G1}\#\mathtt{C1})\#(\mathtt{I2}\#\mathtt{G2}\#\mathtt{C2})) & \\
 & = (\mathtt{I1}\#\mathtt{G1}\#\mathtt{C1}\#\mathtt{I2}\#\mathtt{G2}\#\mathtt{C2}) & \\
 & = ((\mathtt{I1}\#\mathtt{I2})\#(\mathtt{G1}\#\mathtt{G2})\#(\mathtt{C1}\#\mathtt{C2})) & \text{(by rearranging terms)} \\
\multicolumn{3}{l}{\text{But this is of the form:}} \\
\mathtt{T} & = (\mathtt{I}\#\mathtt{G}\#\mathtt{C}) &
\end{array}$$

Closure over Conditional Composition. Conditional composition left-distributes over concurrent and disabling compositions [11].

$$\begin{array}{lll}
(\mathtt{T1} : \mathtt{T2}) & = (\mathtt{T1} : (\mathtt{I2}\#\mathtt{G2}\#\mathtt{C2})) & \\
 & = ((\mathtt{T1} : \mathtt{I2})\#(\mathtt{T1} : \mathtt{G2})\#(\mathtt{T1} : \mathtt{C2})) & \text{(by left-distribution of : over \#)} \\
\multicolumn{3}{l}{\text{But this is of the form:}} \\
\mathtt{T} & = (\mathtt{I}\#\mathtt{G}\#\mathtt{C}) &
\end{array}$$