# Learning Failure Recovery Knowledge for Mechanical Assembly

**L. Seabra Lopes** and **L.M. Camarinha-Matos**

Departamento de Engenharia Electrotécnica, Universidade Nova de Lisboa
Quinta da Torre, P-2825 Monte da Caparica, Portugal

**Abstract.**

*A framework for planning and supervision of robotized assembly tasks is initially presented, with emphasis on failure recovery. The approach to the integration of services and the modeling of tasks, resources and environment is briefly described. A planning strategy and domain knowledge for nominal plan execution is presented. Through the use of machine learning techniques, the supervision architecture will be given capabilities for improving its performance over time. In particular, an approach for memorizing failure recovery episodes, based on abstraction, deductive generalization and feature construction, is presented. Recovery planning consists of adapting plan skeletons from similar episodes previously occurred.*

## 1. Introduction

The concept of Flexible Manufacturing Systems has emerged as an answer to the new challenges that companies around the world are facing, as a consequence of the increasing importance of market niches and of the globalization of the economy. Features like flexibility, adaptability and versatility are desirable in the new generation of automation systems. The efficiency and economical success of such systems depend, however, on the capacity to handle unforeseen events, which occur in a greater number, due to the reduction in structuration constraints. The complexity of flexible manufacturing and assembly processes makes the supervision tasks difficult to perform by humans. Therefore, in manufacturing systems, flexibility and autonomy are tightly related concepts. On-line decision making capabilities have to be included in supervision systems.

### 1.1. Planning and supervision framework

The framework for planning and supervision in manufacturing and assembly systems is illustrated in Fig. 1 (a complete taxonomy of assembly and task planning can be found in [8]). Initially, a *process planning* phase determines all functional constraints (those related to the product, based on geometric model, tolerances model, bill of materials, etc.) and manufacturing constraints (feasibility conditions from technologic point of view, related to equipment that will be used). In which concerns assembly, the generated process plan contains a high level specification of the operations to be performed and respective precedences, goal positions, mating referentials, approaching directions, part grasping zones, part stable poses, resources that can be used in each operation, etc.

When a product order is sent to the shop floor, *execution planning* is performed, which includes shop floor scheduling and detailed execution planning. Scheduling takes into account restrictions imposed by the job size, the order due date, the available resources and the other jobs that will be competing for these resources. Detailed execution planning will determine all actions to be performed in each workcenter, their parameters and exact sequence, therefore completing the executable plan.

Execution planning in the assembly domain did not yet receive enough attention. Until now, the focus has been on joint-level and manipulator-level planning [17] and on the automatic generation of assembly sequences [9]. However, the work on assembly sequence generation, with some exceptions [10], disregards the feasibility constraints imposed by particular assembly cells. An approach for automatically deriving an assembly plan directly executable in a particular assembly cell is presented bellow.

Finally, when the global execution plan for a given order is completed, execution is started. The authors have previously proposed an execution supervision architecture, for flexible assembly systems, having the following main functionalities [1,2]: *Dispatching* of actions to the executing agents, following the scheduled plan; *Execution Monitoring*, i.e. detection of deviations from the nominal system behavior during plan execution; *Failure Diagnosis*, called when the monitoring function detects a deviation,

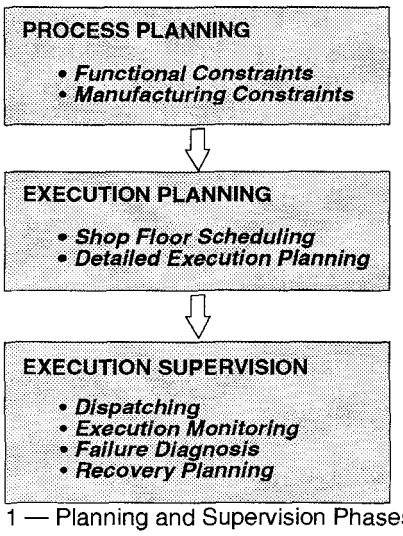


Fig. 1 — Planning and Supervision Phases

Proc. IROS 96
0-7803-3213-X/96/ $ 5.00 © 1996 IEEE

performs failure confirmation, classification and explanation and status identification; and *Failure Recovery*, to determine a recovery strategy to bring execution to a nominal state.

### 1.2. Learning in failure recovery

This paper concentrates on failure recovery. Typically, early approaches to recovery were model-based [7,16,27], avoiding in this way the problem of replanning complexity. The simplest model would be a set of pre-programmed recovery strategies. In [18] pre-programmed strategies were used as scripts and combined with replanning. Donald [3] presented a geometric reasoning approach to error detection and recovery, in which uncertainty is explicitly represented. The SPAR planner is an extension to the classical AI planning paradigm that takes into account uncertainty reduction goals [11]. Unfortunately, building models for complex real-world robotics tasks is difficult or even impossible, and, as Gini pointed out, "the results will be as good as the models used" [6].

The authors have participated in the European ESPRIT project B-LEARN II [12], in which the primary objective was to enhance current robot architectures through the introduction of learning capabilities. The great contribution of learning is to generate models, when a formal model is not available, based on examples. Problems involving uncertainty can often be handled nicely with subsymbolic learning techniques, such as artificial neural networks. In the framework of B-LEARN II and as far as the assembly domain is concerned, learning was used for controller synthesis [19] and for classification tasks in diagnosis [2,22,24,25].

Pre-programming failure recovery strategies presents the same problems as manually synthesing a controller or defining a decision procedure for diagnosis. Again, the integration of learning mechanisms at this level may lead to significant improvements in system autonomy. Learning of planning knowledge in the assembly domain has been attempted with Explanation-Based Learning (EBL) and related techniques. In the ARMS system [26], EBL was used to learn assembly plans based on expert solutions to previous assembly problems. In other approaches [4,28], the same principle is applied to learn recovery strategies. The reported results are promising, but don't seem to be directly applicable, if a complex domain specification is used. It seems that EBL's generalization alone can still produce too specific macro-operators. In this paper we present an approach for learning error recovery knowledge in the assembly domain that combines deductive generalization and abstraction. Failure recovery episodes are memorized and used to guide recovery planning.

## 2. Nominal planning in assembly

### 2.1. Planning strategy

When a given assembly task is selected for execution, the high-level specification contained in the process plan is finally instanciated. It is the moment to plan the execution,

```
 ?- show_frame(assemble_component).

Frame: assemble_component {
isa: tasklevel_operator
op_cost: 1
op_functionality: [
  assemble(R,T,Obj,Comp,Part,Prod,Fix),
  [% Info:
     object_type(Obj,Part),
     part_tool(Part,T),
     component_contacts(Comp,LComp),
     mate(Prod,Comp,Part,_,Prec,_)],
  [% Keep-PC:
     current_tool(R,T),
     not(assembled(Comp,Prod,Fix)),
     fixture_with_product(Fix,Prod),
     not(robot_arm_breakdown(R)),
     not(tool_breakdown(T)),
     not(defective(Obj)),
     all(C,Prec,assembled(C,Prod,Fix)),
     all(C,LComp,
        [ assembled(C,Prod,Fix)),
          represented_by(C,X)]
          -> [not(defective(X))]) ],
  [% Del-PC:
     object_in_robot(Obj,R) ],
  [% Add-C:
     assembled(Comp,Prod,Fix),
     represented_by(Comp,Obj),
     robot_free(R) ] ]
get_op_expansion: get_assemb_expansion_mth
}
Yes

?-
```

Fig. 2 — Planning knowledge: a task-level operator

i.e. to determine all needed actions, their characteristics and parameters and their optimal sequence.

For nominal planning, a planner, in the AI sense of the term, was developed and implemented in Prolog [24]. It uses a domain independent planning strategy, but takes into account domain knowledge provided in a pre-defined format. The planning strategy is, basically, a depth-first forward search procedure. From the initial state of the world, new states are generated, by applying operators of the considered domain, until the goal state is reached. In each step, a set of legal operator instanciations is determined, and evaluated according to domain dependent heuristics. The operator with highest score is selected for continuing the search.

The advantage of the depth-first approach is that, when the heuristics are good, a solution is reached very fast. When that is not the case, the planner may have to spend a lot of time exploring uninteresting alternatives. A new planning strategy, inspired in the well known A* algorithm (best-first search in a problem graph), was implemented. Like the previous strategy, this one is also domain independent. In this case, domain knowledge is specified in three ways. First, frame-based descriptions of the task-level operators (see example in Fig. 2) must be provided. In the description of each operator, the slot `op_functionality`, specifies how the operator affects the world state. It consists of a tuple, OP, with five elements:

`OP = <Op_tmpl, Info, Keep, Del, Add>`

The first item, `Op_tmpl`, is the template of the operator, specifying name and parameters. The second is static information about the domain. The items `Keep` and `Del` are

lists of conditions that must be true before starting execution of the operator (pre-conditions). The conditions in `Keep` are preserved by the operator while those in `Del` are removed. Finally, the item `Add` is a list of conditions that become true as consequence of the execution of the operator. Examples of task-level operators are `assemble` (Fig. 2), `store_tool`, `feed_pallet`, `putdown_object`, `pick_part_from_feeder`, etc., (14 operators in total).

The plan is generated for the nominal execution conditions. The representation of operators contains several assumptions. For instance, one pre-condition of the `assemble` operator is `not(defective(Obj))` (Fig. 2). Moreover, the preconditions of operators can take the form of implications and universally quantified conditions.

The remaining two types of domain knowledge consist of rules to estimate the cost of reaching the goal state starting from a given current state and rules to determine if two planning states are equivalent:

```
cost_estimate(CS,GS)
equivalent_states(S1,S2)
```

It is not possible to completely describe here a realistic planning example. Just for illustration, consider the Cranfield benchmark, a well known laboratory product, used for testing in the assembly domain. It is a pendulum composed of seventeen parts: two side plates, four spacer pegs, a shaft, a lever, a cross bar and eight locking pins. In our experimental setup we have special purpose feeders for side plates and cross bars. The locking pins and the spacer pegs are fed to the system in a pallet and the lever and the shaft in another pallet. Three different tools must be used. Running the planner with the incorporated domain knowledge on this problem finds an optimal plan — 53 operations — in about five minutes. Considering that the average branching factor is 10, this result is acceptable.

### 2.2. Integration in the execution supervisor

The setup being used in our experiments is a robotic assembly cell (Fig. 3), composed of an industrial SCARA robot, three robot grippers, magazine and corresponding tool exchange mechanism, two special purpose feeders and one fixture. As feedback information sources, several discrete information sensors were integrated into the cell. Since, the most frequent execution failures are expected to be those in which the robot arm is involved, including collisions, obstructions and handling failures, a force and torque sensor was also included.

The kernel of the Supervisor is implemented in Prolog, a logic programming language, well known for its powerful term unification procedure, for its inference control strategy and for easy symbol manipulation. Additionally, in order to better structure the main concepts, an in-house developed frame engine, Golog [21], that runs as a shell in Prolog, is used. Golog provides the main features of the frame-based, object-oriented and reactive programming paradigms, namely relations with inclusive and exclusive inheritance, methods and several types of demons. This seems to be the right approach to model both the structural aspects and the

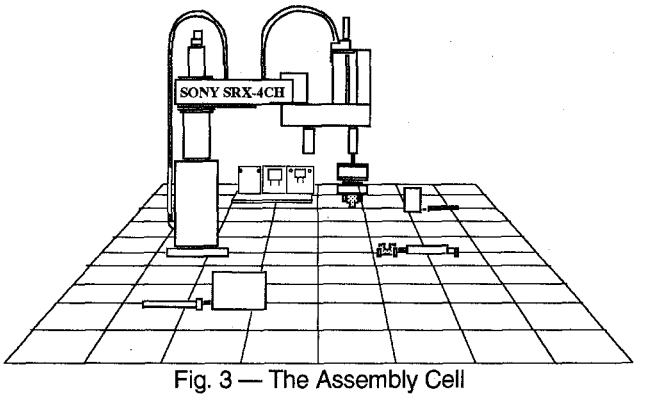

Fig. 3 — The Assembly Cell

dynamic properties of the assembly cell [2]. Most interaction with the hardware is programmed in C language.

The basic structure that integrates the knowledge of the Supervisor is a taxonomy of entities of the domain. The assembly resources are divided in operational resources, sensorial resources and resource storage units. Robots, grippers, feeders, etc., are examples of the first category. In our experimental setup, presence sensors and the force & torque sensor are examples of sensorial resources, while a tool magazine is an example of a resource storage unit.

The assembly resources manipulate artifacts that can fall in three categories: assembly parts, assembled products and unexpected or unknown objects. Assembly parts are used as components of the assembled products. Notice, however, that the expression 'assembly component' is used, not to refer to the physical entity 'assembly part', but to a logical entity that aggregates all the information concerning a given component, namely the type of part that will represent this component, the mating referentials and the relation with other components. In general, a given part type can represent different components in the same product. Each component corresponds to a node in the precedence graph and in the contacts graph of a given product.

Strategic decisions concerning future actions of the system are not taken based on the elementary operations. As was mentioned, planning for nominal execution is performed at a higher level of abstraction, using the so called task-level operators. A generated task plan is a sequence of task operations, each of them represented by a frame in Golog. Each operation inherits, from the corresponding operator, a method that calculates the sequence of elementary operations (i.e. calls to the resource operators) that are necessary to accomplish its goals. This is, basically a compilation or expansion procedure.

## 3. Remembering recovery episodes

Learning is a fundamental ability in intelligent behavior. Understanding and being able to automate and integrate learning mechanisms into machines can dramatically improve the quality of life in our society. In particular, the application of learning techniques to robotics and automation is recognized as a key element for achieving truly flexible manufacturing systems.

Learning control functions, learning classification knowledge and learning planning knowledge are very different tasks requiring different learning paradigms. The attribute-based learning approaches (relying on inductive generalization), which dominated early machine-learning research, are well suited for classification tasks, but of little use in planning. Solving planning problems by observing solutions to similar (previously encountered) problems is, in the first place, a matter of storage and retrieval of cases or episodes and memory organization [14,15,20]. In this section a motivating example is initially presented, and then the approach to the problem of memorizing error recovery episodes is described in detail.

### 3.1. Failure episodes: an example

Defective parts and assemblies are among the external exceptions that may lead to execution failures. Consider, for instance, the situation in which the cross bar (cb1) is going to be assembled to the Cranfield benchmark and the bottom side plate (sp1) is defective (see Fig. 4). The mate operation fails and it is necessary to plan a recovery strategy. In abstract terms, this strategy consists of disassembling the two spacer pegs (peg1 and peg2), substituting the defective side plate by a new part of the same type, re-assembling the pegs and, finally, assembling the cross bar. A recovery strategy, specified in terms of the task-level operators known by the system, is presented in Fig. 5. It is easy to realize that a planner, based on search only, could not find this 24 steps strategy, in reasonable time. With a branching factor of 10, the size of the potential search space for deriving it automatically would be of $10^{24}$ states. Heuristics, which are tailored for nominal planning, would not help.

The first situation of this type that the system encounters can be solved in two different ways: 1) there is some knowledge, heuristics or previous cases that, being tailored for or originating in different situations, still help a powerful planner to calculate the needed recovery strategy in reasonable time; or 2) the human expert provides it. If the recovery strategy is successfully executed, a new error recovery episode is defined.

The authors have previously proposed a learning algorithm, SKIL (structured knowledge by inductive learning [2,22]), that combines inductive generalization and inductive concretion (the opposite of abstraction) and learns a taxonomy of structured anonymous concepts. SKIL can be integrated in the supervision system and used for

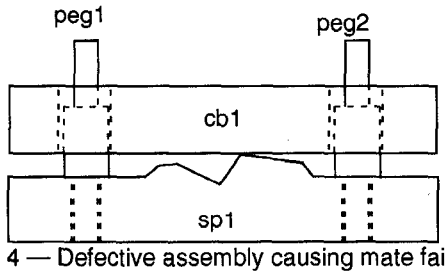

Fig. 4 — Defective assembly causing mate failure

```
putdown(sony,gp_2,o#76)
store_tool(sony,gp_2,tp_2)
get_tool(sony,gp_3,tp_3)
dissassemble(sony,gp_3,p1,peg2,peg,cranf,fix_1)
putdown(sony,gp_3,peg_o1)
dissassemble(sony,gp_3,p2,peg1,peg,cranf,fix_1)
putdown(sony,gp_3,peg_o2)
store_tool(sony,gp_3,tp_3)
get_tool(sony,gp_1,tp_1)
dissassemble(sony,gp_1,o#66,sp1,sp,cranf,fix_1)
putdown(sony_scara,gp_1,o#66)
feed_part(feeder_sp,o#99,side_plate)
pick_part_from_feeder(sony,gp_1,o#99,sp,fd_sp)
assemble(sony,gp_1,o#99,sp1,sp,cranf,fix_1)
store_tool(sony_scara,gp_1,toolp_1)
get_tool(sony_scara,gp_3,toolp_3)
pickup_part(sony_scara,gp_3,p1,peg)
assemble(sony,gp_3,p1,peg2,peg,cranf,fix_1)
pickup_part(sony_scara,gp_3,p2,peg)
assemble(sony,gp_3,p2,peg1,peg,cranf,fix_1)
store_tool(sony_scara,gp_3,toolp_3)
get_tool(sony_scara,gp_2,toolp_2)
pickup_part(sony,gp_2,o#76,cb)
assemble(sony,gp_2,o#76,cb1,cb,cranf,fix_1)
```
Fig. 5 — The recovery plan for the situation in Fig. 4

diagnosis. A post-processing phase must be added in order to make explicit the state update information, already present in the learned failure descriptions. In the end, a failure diagnosis will be composed of the diagnosis template as well as the template of the operation that failed (both for matching) and the add and delete lists, for state update. Add and delete lists assume that none of the effects of the failed operation actually took place.

For instance, the failure in Fig. 4, would fall into the following general diagnosis:

```
failure_diagnosis(
   defective_assembly(Obj_1,Comp_1,Prod,Fix),
   assemble(R,T,Obj_2,Comp_2,Part,Prod,Fix),
   [ ], % delete list
   [ defective(Obj_1) ] % add list
).
```

In this particular case, Comp_1 would come instanciated to sp1, Comp_2 would come instanciated to cb1, etc.

Finally, a failure recovery episode, Ep, is a triple with the following form:

Ep = <FOp, FDg, RPlan>

where FOp is the template of the failed operation, FDg is the template of the failure diagnosis description and RPlan is the recovery plan.

### 3.2. Episode generalization

The first processing step, applied to a failure recovery episode, before storage, is a deductive generalization transformation. The idea of generalizing a plan for future use was first introduced in the early STRIPS/PLANEX planning and execution system [5]. Each generalized plan would constitute a new macro-operator. The applied generalization procedure is the root of what became known

later as Explanation-Based Learning (EBL) [13]. In EBL, acquiring a concept relies on using a domain theory to analyze why a previous solution was or was not successful.

In our system, the generalization procedure takes inspiration in PLANEX and EBL. The state of the cell, immediately after the failure (in short, the failure state), must be calculated and all information facts, needed to prove the plan, must be gathered. These lists of conditions and information facts are initialized according to the operation that failed and the definition of the failure diagnosis. Then the plan is traversed. As each operator in the plan is applied, the current lists of information facts and initial conditions are updated, according to the pre-conditions imposed by that operator and the information facts used. In this process, the successive states of the system are generated, by applying the delete lists and add lists of the operators. A pre-condition is added to the list of initial conditions (which describe the failure state) only if it cannot be proved in the current state.

Generalization, in this case, is the deduction of an error recovery episode that, if adequately instanciated with constants, becomes the original episode. To perform this generalization, all constants in the failure, in the list of information facts and in the episode description itself (including failed operation, failure diagnosis and recovery plan) are substituted by variables. At this point, each occurrence of a constant generated a different variable.

Then, a proof is made that, using the generalized information facts, the generalized recovery plan is applicable in the generalized failure state. However, there are many possible proof trees. Since the proof tree, that is convenient for generalization, is the exact mirror of the proof tree of the original plan, both proofs are conducted in parallel. The proof of the instanciated plan guides the proof of the generalized plan. During the proof, restrictions are imposed on variables. Some variables are substituted for other variables in the plan, so that each constant in the original episode, wherever it appears playing the same role, will be represented by one and only one variable. The unification mechanism of Prolog greatly simplifies this task.

### 3.3. Episode abstraction

The knowledge produced by deductive generalization is often too specific to be useful in the future. For instance, a generalization of a plan to assemble a few components, is of little use. Each component has a different set of precedence relations with other components. In generalization, each preceding component is substituted for a variable, but the precedence relation itself is preserved. This leads to a too specific generalization. Too specific knowledge is expensive both in terms of storage space and retrieval time.

Humans don't keep too many details in memory. Human remembering is often the process of reconstructing what must have happened rather than directly retrieving what did happen [14,20]. Applying abstraction mechanisms in order to reduce the level of detail in an episode description seems to be the right approach. Abstraction should not be confused with generalization. While abstracting a piece of knowledge reduces the level of detail in the description, generalizing it enlarges the set of entities to which it applies.

```
assemble(Obj,Comp)
• assemble(R,T,Obj,Comp,Part,Prod,Fix)

disassemble(Obj,Comp)
• disassemble(R,T,Obj,Comp,Part,Prod,Fix)

pick(Obj)
• pickup_object(R,T,Obj)
• pickup_part(R,T,Obj,Part)
• pick_part_from_pallet(R,T,Obj,Part,Pal,PSys)
• pick_part_from_feeder(R,T,Obj,Part,Fd)

place(Obj)
• putdown_object(R,T,Obj)
• place_part_in_pallet(R,T,Obj,Part,Pal,PSys)

nil
• feed_pallet(Pal,PSys)
• putaway_pallet(Pal,PSys)
• get_tool(R,T,TP)
• store_tool(R,T,TP)
• set_fixture_for_product(Fix,Prod)
```

Fig. 6 — Taxonomy of operators

The particular approach that was followed in this research was to use a taxonomy of operators as the reference structure for abstraction (see Fig. 6). Each operation in a plan is abstracted according to the taxonomy. This means that some operations and some parameters in other operations will disappear. The abstract operators have no consistent definition, and so the abstract plan cannot be proved. It can, however, be used as a guide in replanning.

### 3.4. Features and clusters

Features or attributes are used to describe examples in inductive learning and pattern recognition and to index and retrieve cases in case-based reasoning [15]. Constructed features make explicit relationships that would otherwise be implicit in the input. A feature construction (or feature extraction) phase is usually applied to the raw data (e.g. measurements in a physical system) before problem solving [25]. In the assembly application of this paper, when a failure is detected, a failure recovery strategy will be determined (if possible) based on similar failure recovery episodes, previously occurred and memorized. In the current approach, indexing memorized episodes does not require any feature construction. The descriptions of the failure and the failed operation constitute the indexing key. However, features are still used in two ways: 1) to assess similarity between a reconstructed plan and a memorized plan skeleton; and 2) to define plan clusters in an abstract recovery plan, before storage of the complete episode.

The entities appearing in the indexing key will partially instanciate the retrieved plan skeleton. The instanciated skeleton is then used to guide the search for the plan to recover from the particular situation at hand. This is basically a plan adaptation phase, during which additional variables are instanciated and the specific operations determined.

After the two transformations previously described (generalization and abstraction), features are constructed for all entities appearing in the plan but not appearing in the

indexing key (i.e. the failed operation and the failure description). In the current approach, a feature of an entity is a relationship between that entity and another entity which appears in the indexing key. For instance, suppose that, after generalizing the plan in fig. 5, the components 'sp1' and 'peg1' are represented by variables SP1 and PEG1. SP1 appears in the generalized failure description and PEG1 appears in the generalized plan, but not in the generalized operation template or generalized failure description. A feature of PEG1 would be:

```
assemble_after(PEG1, SP1)
```

As in other learning approaches, feature construction strategies must be defined by the human operator. New feature definitions can be added to the system at any time without affecting what was previously learned. These features will be used to choose the best instanciations of the plan skeleton during plan adaptation.

Features are also used to define clusters in a plan skeleton. In this context, a cluster represents a part of a plan where a pattern repeats several times. For instance, disassembling and puting components on the table, or picking up and assembling them, are two patterns that occur repeatedly in the plan of Fig. 5. There are many failure situations similar to this one, with the only main difference that a greater or a smaller number of components must be disassembled and then re-assembled. It would be nice if the learned plan skeleton could be used to derive recovery plans with a different number of repetitions of those patterns.

There are several possible approaches to the formation of these clusters. On one extreme, only one cluster would be defined, incorporating all consecutive occurrences of the pattern, and only the features common to all these occurrences would be stored. On the other extreme, each occurrence of the pattern would define a different cluster. An intermediate approach was preferred. In this approach, which received the name of *logarithmic clustering*, the number of defined clusters will be greater or equal to K, according to the following expression:

$$K = \log_2 R$$

where R is the number of repetitions of the pattern. Clustering is as follows:

1. **Define** a cluster having all repetitions of the pattern and all features common to these repetitions.
2. **If** the current number of clusters is greater or equal to K, **stop**.
3. **For each** cluster, CLUSTER, currently defined do:
   3.1. **Determine** the feature or features that appear in the longest sub-sequence of pattern repetitions in CLUSTER.
   3.2. **Define** new clusters, one for the sub-sequence that was found, and one or two more for the remaining portions of CLUSTER not included in that sequence.
   3.3. **Remove** CLUSTER.
4. **Goto** 2.

This procedure is applied to all parts of the previously generalized and abstracted plan where repeating patterns can be detected. Two sequences of operations are considered to belong to the same pattern if the following conditions hold: a) the names and order of the operations are the same; b) the variables representing entities mentioned in the indexing key (failed operation and failure description) are the same and play the same role; and c) to each variable playing a given role in a sequence, corresponds another variable playing that role in the other sequence.

Applying all these transformations to the recovery plan in Fig. 5 will produce the plan skeleton presented in Fig. 7 (variables internal do the plan skeleton are identified in the Prolog syntax, i.e. underscore followed by a number). The initial disassembly phase became a cluster. The presented features describe the relationships between the components that are disassembled in this cluster and entities in the indexing key, namely SP1 and CB1. The assembly phase, after substituting the defective component, became another cluster in the plan skeleton.

```
?- failure_recovery_episode(
      assemble(R,T,Obj_cb1,CB1,CB,Cranf,Fix),
      defective_assembly(Obj1_sp1,SP1,Cranf,Fix),
      Plan_skeleton
   ).
....
Plan_skeleton = [
   [ place(Obj_cb1), [ ] ]
   [ [ disassemble(_10,_20), place(_10) ],
         [ initially_in_contact(_10,Obj1_sp1),
           assemble_after(_20,SP1),
           assemble_before(_20,CB1),
           in_contact_with(_20,SP1),
           in_contact_with(_20,CB1) ] ]
   [ disassemble(Obj1_sp1,SP1), [ ] ]
   [ place(Obj1_sp1), [ ] ]
   [ pick(_30),
         [ object_type(_30,SP) ] ]
   [ assemble(_30,SP1),
         [ object_type(_30,SP) ] ]
   [ [ pick(_40), assemble(_40,_50) ],
         [ initially_in_contact(_40,Obj1_sp1),
           assemble_after(_50,SP1),
           assemble_before(_50,CB1),
           in_contact_with(_50,SP1),
           in_contact_with(_50,CB1) ] ]
   [ pick(Obj_cb1), [ ] ]
   [ assemble(Obj_cb1,CB1), [ ] ]
]
yes
```

Fig. 7 — The plan skeleton in the stored recovery episode of Figs. 4 and 5.

## 4. Failure recovery planning

In their stored form, error recovery episodes can be thought of as conceptual categories, but not in the usual sense of the word. Although these categories organize similar concepts, their members cannot be enumerated. They can, however, be reconstructed as necessary.

In the supervision system, when a failure is detected and the diagnosis function is able to produce the failure description, recovery planning is attempted. The first step is to look for similar failure situations previously encountered. Using the failed operation and the failure description as indexing key, an error recovery episode will be retrieved. Then, the plan skeleton in that episode is

adapted to solve the current situation. This adaptation is, basically, a state-based planning procedure, similar to the nominal planner briefly described in section 2.1, but instead of using heuristics, the plan skeleton is used as a guide. Its structure is as follows:

**Let** FOp be the failed operation.

**Let** FDg be the failure diagnosis description.

**Let** Ep = <FOp, FDg, Plan_sklt> be the retrieved episode, already partially instanciated with the constants contained in FOp and FDg. Plan_sklt is the plan skeleton, that is used to guide replanning.

**Let** FSt be the failure state.

**Let** GSt be the goal state, in which the system can resume the nominal plan.

1. **Define** a node in the search space, Nd, corresponding to the failure state and having an empty plan and the complete plan skeleton to traverse:
   Nd = <FSt, [ ], Plan_sklt>
2. **Select** the most promissing Nd = <St, Plan, Sklt> for expansion. This selection is made based on the following three criteria, in descending order of priority:
   a) The part of the plan skeleton, that still has to be traversed, Sklt, has the smallest lenght found.
   b) The currently built plan, Plan, has the smallest lenght.
   c) The currently built plan, Plan, is the most similar to the part of the plan skeleton that was already consumed ( Plan_sklt - Sklt ). This similarity is evaluated in terms of the proportion of features in the skeleton that are verified in Plan.
3. **If** skeleton Sklt is empty, **goto** 5.
4. **Let** Step be the first element in Sklt.
   4.1. Clusters can be skipped. **If** Step is a cluster, **define** a new node N = <St, Plan, Sklt_1>, in which Sklt_1 is the rest of Sklt.
   4.2. **If** Step is a cluster, **expand** it to a sequence of abstract operations, SAOp, and **define** Sklt_2 to be the concatenation of SAOp to Sklt.
   **Otherwise define** Sklt_2 to be equal to Sklt.
   4.3. **Let** AOp be the first abstract operation in Sklt_2.
   **Let** Sklt_3 be the remaining steps in Sklt_2.
   **Generate** all applications of task-level operators, which belong to the class of AOp (according to the taxonomy of Fig. 9), in state St.
5. **Generate** also all applications of task-level operators, which belong to the nil class in the taxonomy of Fig. 9, in state St.
6. **For each** task-level operator application, TAOp, from all those generated in steps 4 and 5, do:
   6.1. Calculate the new state, NSt, and append TAOp to Plan, to obtain the new plan, NPlan.
   6.2. **If** NSt is equivalent to the goal state, GSt, **Return** NPlan.
   **Otherwise define** a new node:
   N = <NSt, NPlan, NSklt>
   (the new skeleton, NSklt, will be Sklt_3 if the operator application was generated in step 4 or Sklt_2 if it was generated in step 5).
7. **Goto** 2.

In this description of the recovery planning algorithm, several details are omitted, for instance the removal of redundant states.

Although this approach to recovery planning seems very promising, extensive evaluation still has to be performed. The first experiments show that recovery plan skeletons, stored in the failure recovery episodes, are a good guide in recovery planning, enabling to solve problems that, otherwise, would be solvable only with help of very good pre-programmed heuristics.

## 5 Conclusions

On-line decision making capabilities must be included in manufacturing systems in order to comply with the new requirements of flexibility and autonomy. In previous papers [1,2,23], the authors have proposed an architecture for supervision and programming by demonstration in the domain of robotized assembly tasks. The main required functionalities were identified, namely nominal execution planning, system training, operations dispatching, execution monitoring, failure diagnosis and failure recovery. The system training module coordinates the interaction of the human operator with assembly system, in order to facilitate the acquisition of monitoring, diagnosis and recovery knowledge. The programming by demonstration paradigm plays an important role in the architecture.

In this paper, developments in planning for nominal execution were first presented. The used planning strategy is forward chaining and based on the A* search algorithm and pre-programmed heuristics. As this approach cannot be applied to recovery planning, due to the difficulty in programming heuristics for all the failure situations that might occur, the programming by demonstration paradigm is extended to address failure recovery. In this case, it is expected that the human operator will provide a failure recovery strategy whenever the system cannot find one automatically. If this strategy is successful, the error recovery episode is stored in a form that preserves the essential informations but not the details. To achieve this representation, abstraction and deductive generalization mechanisms are applied. Clustering of repeating patterns in recovery plans is also performed. In future system performance, when similar situations arise, these episodes are indexed, using information on the failed operation and on the failure diagnosis, and retrieved, and the recovery strategies that were found successful in the past are adapted for the new situations.

The described approach is only intended for high-level failure recovery planning. It requires an execution infrastructure providing the elementary operations of the

system. At this lower level, reactive planning and control play the main role. The B-LEARN II project investigated the application of subsymbolic learning techniques (including different types of neural nets) and reinforcement learning schemes to improve control and coordination in robotics [12]. High-level recovery planning also needs qualitative representations of the world and the detected failures. These representations must be built and maintained by monitoring and diagnosis activities, based on sensors which deliver a rich amount of parameters of different nature and reliability. Our project investigated the application of learning in monitoring and diagnosis [2, 22]. Typically, the human operator provides classified examples of normal and abnormal behavior (obtained by directly executing the tasks and observing system behavior) and the learning module, using inductive techniques, generates a hierarchical failure monitoring and diagnosis model.

## Acknowledgments

This work has been funded in part by the European Community (ESPRIT projects B-LEARN II and FlexSys) and JNICT (project CIM-CASE).

## References

[1] Camarinha-Matos, L.M., L. Seabra Lopes, J. Barata (1994) Execution Monitoring in Assembly with Learning Capabilities, *Proc. of IEEE Int'l Conf. on Robotics and Automation*, San Diego.

[2] Camarinha-Matos, L.M.; Seabra Lopes, L.; Barata, J. (1996) Integration and Learning in Supervision of Flexible Assembly Systems, *IEEE Transactions on Robotics and Automation* (to appear).

[3] B.R. Donald (1988) A Geometric Approach to Error Recovery for Robot Motion Planning with Uncertainty, *Artificial Intelligence*, 37, pp. 223-271.

[4] E.Z. Evans, C.S. George Lee (1994) Automatic Generation of Error Recovery Knowledge Through Learned Reactivity, *Proc. IEEE Int'l Conf. on Robotics and Automation*, San Diego, California, v. 4, pp. 2915-2920.

[5] R.E. Fikes, P.E.Hart, N.J. Nilsson (1972) Learning and Executing Generalized Robot Plans, *Artificial Intelligence*, 3 (4), pp. 251-288.

[6] M. Gini (1987) Symbolic and Qualitative Reasoning for Error Recovery in Robot Programs, *NATO ASI Series, Lang. for Sensor-Based Control*, vol. F29.

[7] M. Gini, G. Gini (1983) Towards Automatic Error Recovery in Robot Programs, *Proc. Int'l Joint Conf. on Artificial Intelligence*, Karlsruhe, Germany, pp. 821-823.

[8] S. Gottschlich, C. Ramos, D. Lyons (1994) Assembly and Task Planning: a Taxonomy, *IEEE Robotics and Automation Magazine*, vol. 1 (3), pp. 4-12.

[9] L.S. Homem de Mello, S. Lee [ed.] (1991) *Computer-Aided Mechanical Assembly Planning*, Kluwer Academic Publishers, Boston.

[10] Y.F. Huang, C.S.G. Lee (1990) An Automatic Assembly Planning System, *Proc. IEEE Int'l Conf. on Robotics and Automation*, Cincinnati, Ohio, pp. 1594-1599.

[11] S. A. Hutchinson, A. C. Kak (1990) SPAR: A Planner that Satisfies Operational and Geometric Goals in Uncertain Environments, *AI Magazine*, vol. 11, pp. 30-61.

[12] M. Kaiser, A. Giordana, M. Nuttin, L. Seabra Lopes [ed.] (1995) *B-LEARN II: Combining Sensing and Action* (Final Project Report), ESPRIT BRA 7274, Karlsruhe.

[13] S.T. Kedar-Cabelli, L.T. McCarty (1987) Explanation-Based Generalization as Resolution Theorem Proving, *Proc. 4th Int'l Workshop on Machine Learning*, Irvine, CA, pp. 383-389.

[14] J.L. Kolodner (1984) *Retrieval and Organizational Strategies in Conceptual Memory: a Computer Model*, Lawrence Erlbaum Associates Publishers, Hillsdale, New Jersey.

[15] J.L. Kolodner (1993) *Case-Based Reasoning*, Morgan Kaufman Publ., San Mateo.

[16] E. López-Mellado and R. Alami (1990) A Failure Recovery Scheme for Assembly Workcells, *Proc. IEEE Int. Conf. on Robotics and Automation*, Cincinnati, pp. 702-707.

[17] P.J. McKerrow (1991) *Introduction to Robotics*, Addison-Wesley Pub. Co., Sydney.

[18] G.R. Meijer (1991) *Autonomous Shopfloor Systems. A Study into Exception Handling for Robot Control* (PhD Thesis), Universiteit van Amsterdam, Amsterdam.

[19] M. Nuttin, H. Van Brussel (1995) Learning an Industrial Assembly Task with Complex Objects and Tolerances, *Proc. 4th European Workshop on Learning Robots*, Karlsruhe, Germany, pp. 27-37, December 1995.

[20] R.G. Schank (1982) *Dynamic Memory. A Theory of Reminding and Learning in Computers and People*, Cambridge University Press, Cambridge.

[21] Seabra Lopes, L. (1994) *Golog 2.0. Um Gestor de Objectos em Prolog* (in Portuguese), Technical Report, UNL-12-94.

[22] Seabra Lopes, L.; Camarinha-Matos, L.M (1995) Inductive Generation of Diagnostic Knowledge for Autonomous Assembly, *Proc. IEEE Int'l Conf. on Robotics and Automation*, Nagoya, JP.

[23] Seabra Lopes, L.; Camarinha-Matos, L.M (1995) Planning, Training and Learning in Supervision of Flexible Assembly Systems, *Balanced Automation Systems*, IFIP / Chapman & Hall, pp. 63-74.

[24] Seabra Lopes, L.; Camarinha-Matos, L.M. (1995) A Machine Learning Approach to Error Detection and Recovery in Assembly, *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*.

[25] L. Seabra Lopes, L.M. Camarinha-Matos (1995) Example Generation and Processing for Inductive Learning in the Assembly Domain, *Proc. 4th European Workshop on Learning Robots*, Karlsruhe, Germany, pp. 5-13.

[26] A.M. Segre (1988) *Machine Learning of Robot Assembly Plans*, Kluwer Academic Publishers, Boston.

[27] S. Srinivas (1976) *Error Recovery in a Robot System* (PhD Thesis), California Institute of Technology.

[28] Y. Zheng and L.K. Daneshmend (1991) Learning Error-Recovery Strategies in Telerobotic Systems, *Proc. 1991 IEEE International Conference on Robotics and Automation*, Sacramento, California, 1, pp. 252-259.