A Machine Learning Approach to Error Detection and Recovery in 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.

Research results concerning error detection and recovery in robotized assembly systems, key components of flexible manufacturing systems, are presented. A planning strategy and domain knowledge for nominal plan execution and for error recovery is described. A supervision architecture provides, at different levels of abstraction, functions for dispatching actions, monitoring their execution, and diagnosing and recovering from failures. Through the use of machine learning techniques, the supervision architecture will be given capabilities for improving its performance over time. Particular attention is given to the inductive generation of structured classification knowledge for diagnosis.

1 Introduction

The increasing globalization of the economy and the emergence of economic blocks is imposing tough challenges to manufacturing companies. The ability to produce highly customized products, in order to satisfy market niches, requires the introduction of new features in automation systems — flexibility, adaptability, versatility — relaxing cell structuration constraints and leading to the concept of Flexible Manufacturing 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 processes makes the supervision and maintenance 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.

Research results concerning execution supervision in flexible assembly systems (major components of today's manufacturing systems) are presented bellow. An architecture is proposed that considers, at different levels of abstraction, functions for dispatching actions, monitoring their execution, and diagnosing and recovering from failures. One main problem is the acquisition of knowledge about the task and the environment to support monitoring, diagnosis and recovery. For this purpose, the use of machine learning techniques is being investigated, in the context of the ESPRIT project B-LEARN II.

2 Planning, Supervision and Learning

In the lower levels of the hierarchy of enterprise activities, several relevant planning procedures are executed, namely product and process oriented planning, scheduling at the shop floor and detailed execution planning.

Product oriented planning, based on the product model (bill of materials, geometric model, etc.), determines a feasible assembly precedence graph. Process oriented planning, taking into account feasibility conditions from the technologic point of view, generates additional constraints, more concerned with the equipment that must be used. The two phases of product oriented planning and process oriented planning can more easily be performed in interaction with the humans. The process plan typically includes information on abstract operations to be performed and respective precedences, goal positions, mating referentials, approaching directions, possible part grasping zones, part stable poses, types of resources that can be used in each operation, etc.

When a specific cell is selected to perform an assembly task, the high-level specification contained in the process plan is finally instantiated. It is then the moment to plan the execution, i.e. to determine all needed actions, their characteristics and parameters and their optimal sequence. For nominal planning, and also for failure recovery, a planner, in the AI sense of the term, was developed [10]. In this section, the planning strategy, the representation issues, the domain knowledge and the supervision functions are presented.

2.1 Planning Strategy

The planner, implemented in Prolog, 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 instantiations is determined, and evaluated according to domain dependent heuristics. The operator with highest score is selected for continuing the search. The way the planner uses the provided domain knowledge to select

0-8186-7108-4/95 $4.00

operators makes it a non-linear planner, since it can handle interaction between goals.

The domain specification includes: definition of operators; typical precedences between facts in the goal state; and measures of contribution of facts asserted in a given phase to goal facts. An operator is defined as a Prolog clause whose head is:

```
operator(Op,Info,Keep,Del,Add)
```

The first argument, `Op`, is the template of the operator, specifying name and parameters. The second is static information about the domain. The arguments `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 argument `Add` is a list of conditions that become true as consequence of the execution of the operator.

One of the ways to handle non-linear planning problems is to incorporate in the planning process knowledge about goal interaction. In many domains it is quite simple to provide or determine precedence relations between facts in the goal state. The planning system, being described, can take into account such precedence relations. The goals not preceded by any other goals in the goal state are selected to be solved first (therefore, an instance of the least-commitment strategy [11] is used). When each of them is solved, some new goals will be considered by the planner, and so on. In the blocks world, the following precedence rule would be enough:

```
typical_precedence(on(B,C),on(A,B)).
```

In each phase of the planning process, the pre-conditions of all operators will be matched to the current world state. The result is a set of legal operator instantiations whose usefulness, concerning the solution of the goals currently being considered, is evaluated. This evaluation is made taking into account the contributions of each of the added facts to each of the goals currently being considered. This evaluation is, again, domain dependent, and is made based on Prolog rules of the following form:

```
contrib_to_goal(CS,F,G,Score):- ... .
```

Basically, the rule says that in a given state CS, an operation that asserts the fact F will contribute to goal G in a way measured by the returned Score.

2.2 The Assembly Domain Knowledge

In assembly, domain knowledge is much more complex than in the blocks world. At the center of the problem are the graphs of precedences for assembly and for disassembly and the graph of connections between components. The planning strategy, just described, is used to determine sequences of actions, at a sufficiently high level of abstraction. Still, the planner must take into account the positions of parts in feeders and pallets, the mating positions, the mating precedences, the tools for grasping and mating, the needed elementary skills, etc. Examples of an operator, a precedence rule and a contribution rule are

```
operator(
  assemble_component
     (R,T,Obj,Comp,Part,Prod,Fix,Cp,Geom),
  [% Info:
     object_type(Obj,Part),
     part_tool(Part,T),
     component_contacts(Comp,LComp),
     mate(Prod,Comp,Part,Geom,Prec,Succ)],
  [% 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:
     current_arm_position(R,Cp),
     object_in_robot(Obj,R) ],
  [% Add-C:
     assembled(Comp,Prod,Fix),
     represented_by(Comp,Obj),
     robot_free(R),
     current_arm_position(R,Dp) ]
).

typical_precedence(
        assembled(C1,P,F),
        assembled(C2,P,F))
:- clause(mate(P,C2,_,_,Prec,_),true),
   member(C1,Prec).

contrib_to_goal(CS,
  pallet_available(Pal,_),
  assembled(C,Pd,_),
  3) :- member(part_in_pallet(Ob,Pal),CS),
        member(object_type(Ob,Part),CS),
        member(mate(Pd,C,Part,_,LP,_),CS),
        check_prec(CS,C,Pd,LP).
```

Fig. 1 — Assembly Planning Knowledge

presented in Fig. 1. Besides the shown `assemble` operator, the planner can make use of a `disassemble` operator, especially useful in correcting assembly errors, and various operators for picking and placing parts in/from fixtures, feeders or the free workspace, for fetching and storing tools, for feeding parts and pallets, for locating objects, etc.

It is not possible to completely describe here a realistic planning example. Just for illustration, we mention an experiment with 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. Except for the locking pins, all the other needed mate operations are stack operations, the most common in industrial practice. All mate operations require

some degree of compliance. Compliance, however is not handled at the level of action sequence planning. 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 produces an optimal plan with 53 operations without any backtracking, which may be considered a good result.

Each of the operations in the generated plan must be expanded, in a more or less deterministic way, into a sequence of resource-level operators, like `move`, `approach`, `transfer`, `peg_in_hole` or `grasp`. This involves lower levels of planning, more concerned, for instance, with trajectories or compliance. A skill acquisition approach to compliance, using learning techniques, seems promising [4]. However, as we are more concerned with execution supervision, we prefer to simplify the planning functionalities of the lower levels in order to be able to realize a working prototype.

2.3 Supervision Functions

Two plan levels were already mentioned. A hierarchical specification of the mate precedence graph or the learning of macro-operations may originate additional plan levels. A hierarchical plan is an advantage for supervision, since it provides different contexts for error detection and recovery. At the lower levels, error recovery will tend to consist of simple reflexive actions. At the upper levels recovery will require more extensive diagnosis and planning. The architecture of an Intelligent Execution Supervisor should reflect the hierarchical structure of the plans. At each plan level, the main functions are [1]:

Dispatching and Global Coordination — The global coordination activities performed by a high level controller include dispatching actions to the executing agents, driven by the scheduled task plan, synchronization of the agents activities and with external events, and information sharing.

Monitoring of Assembly Plans — The monitoring function is used to detect off-nominal feedback in the system during plan execution (i.e. deviations from normal behavior).

Failure Diagnosis — When the monitoring function detects a deviation the diagnosis function is called. In general, diagnosis consists of four steps: failure confirmation, failure classification, failure explanation and status identification. At each execution level, different degrees of explanation for a failure may be generated, depending on the information available.

Failure Recovery — At each supervision level, the recovery function is called when the diagnosis function confirmed a failure and found an explanation. The goal is to determine a recovery strategy to bring the execution to a nominal state.

2.4 Training and Learning

In real execution, a feature extraction function is permanently acquiring monitoring features from the raw sensor data. The monitoring function compares these features with the nominal action behavior model. For example, let's consider that, during the execution of a `Transfer` operation, in which the robot carries a part to be assembled, an object, unexpectedly originating in the environment, collides with the gripper. The first diagram, included in Fig. 2, shows the perceived sensor data during failure occurrence. The second diagram shows a qualitative model of the operation. The third diagram shows a qualitative interpretation of the raw sensor data in terms of the features used in the operation model. Since a deviation is detected, the diagnosis function is called to verify if an execution failure occurred and, that being the case, determine a failure classification and explanation. For this function, additional features must be extracted. Diagnosis is a decision procedure that requires a sophisticated model of the task, the system and the environment. The final step, based on the failure characterization, is recovery planning.

The problem of building the knowledge base, and in particular the models that the monitoring, diagnosis and recovery functions need, is not easily solved. Even the best domain expert will have difficulty in specifying the necessary mappings between the available sensors on one side and the monitoring conditions, failure classifications, failure explanations and recovery strategies on the other. Also, a few less common errors will be forgotten. Known prototype systems show limited domain knowledge, as they are intended mainly for exemplification and not to be used as robust solutions in the real world. Thus, we include in the execution supervisor two other functions: *training* and *learning* (Fig. 3). The training module coordinates the interaction with the human operator in order to acquire new information about nominal execution of the assembly plans as well as descriptions of new error situations. The learning module compiles raw data generating classification knowledge, generalizes instances of target concepts, etc., in order to build the needed models.

According to the paradigm of Programming by Human Demonstration (in this case, robot programming by demonstration [4,11]), complex systems are programmed by showing particular examples of their desired behavior and giving explanations for particular failure situations. In our current approach, the interaction between the execution supervisor and the human operator is fundamental. The human will carry out an initial training phase for the nominal plan execution. The traces of all testable sensors will be collected during training in order to generate the

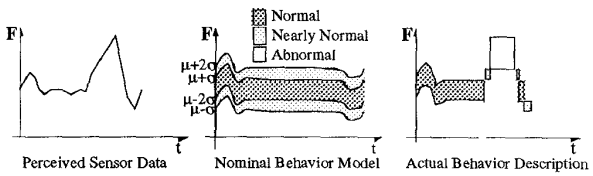

Fig. 2 — Qualitative Features for Monitoring

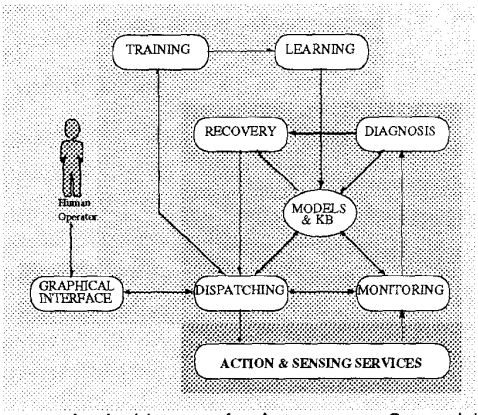

Fig. 3 — An Architecture for Autonomous Supervision

corresponding monitoring knowledge.

In the existing implementation, for each action and each continuous feature, its typical behavior during the execution of the action is calculated as being the region between the average minus standard deviation behavior and the average plus standard deviation behavior (Fig. 2). The trace of discrete features is also recorded. Also in the initial training phase, the human operator may decide to provoke typical errors, in order to collect raw data in error situations. Error classification knowledge is subsequently generated by induction. When a new failure is detected during real execution of the assembly system, the human operator is called to classify and explain that failure and to provide a recovery strategy for the situation. This is considered also as a training action, since the system history and the model of errors will be expanded and new knowledge will eventually be generated by incremental induction, therefore improving future system performance.

2.5 Experimental Setup

The setup being used in our experiments is a robotic assembly cell, 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.

3 Failure Diagnosis

A qualitative approach to modeling errors was followed in our work. Depending on the available sensor information, a more or less detailed classification and explanation for the detected execution failure may be obtained. Therefore, the model of errors should be hierarchical or taxonomic. At each level of the taxonomy, cause-effect relations between different types of errors should be added. Typically, execution failures are caused by system faults, external exceptions or other past execution failures, although, in general, errors of the three kinds may cause each other [1]. Determining explanations for detected execution failures can become very complex when errors propagate. Modeling errors in terms of taxonomic and causal links aims at handling this complexity [1,6].

3.1 Learning of Conceptual Hierarchies

As emphasized above, the difficulty in hand-coding the models that the supervision functions need, raises the question of how to build such models automatically. The classification phase of the diagnosis task can be performed based on knowledge generated by induction.

In a previous phase of this research, several inductive learning algorithms and systems were applied to the assembly domain [1,9]. These techniques are only able to learn uni-dimensional concept descriptions: the resulting knowledge is only able to assign classes to objects from a given domain. In the assembly domain, for example, these algorithms and systems cannot handle the problem of discriminating collisions from obstructions and normal situations, handling simultaneously the problem of discriminating between different types of collisions.

Having as motivation the automatic construction of the models required for the assembly supervisor, the idea of generating a concept hierarchy became attractive. The problem of learning at multiple levels of abstraction has not yet been adequately considered in the literature. In some approaches, a fixed decomposition of concepts is used, and learning is applied at each level [6]. However this is not flexible enough. Fixed decompositions have also been used for feature values [6,8]. A new algorithm, SKIL (structured knowledge generated by inductive learning), was developed to perform this task [9]. The concepts in the hierarchy learned by SKIL are characterized by a set of symbolic classification attributes.

At the lower levels of the hierarchy, concepts are described in more detail, i.e., more attribute values are specified. Moreover, in detailing or refining a concept, in which attributes take certain values, it may make sense to calculate other attributes. Therefore, the user should provide a set of attribute enabling statements of the form (A_i, A_{ij}, EA_{ij}), meaning that when the value of A_i is determined to be A_{ij}, then attributes in EA_{ij} should be included in the set of attributes to consider in the continuation of the induction process. For example, when learning the behavior of a `Transfer` operation, if a collision is found, it may make sense to determine some characteristics of the colliding object, like size, hardness and weight. This could be expressed by the following attribute enabling triple:

```
(failure_type, collision, {obj_size,
    obj_hardness, obj_weight})
```

The attribute values of the concepts in the hierarchy are determined inductively based on training data specified in terms of a set of discrimination attributes or features, which can be numerical or symbolic. Each example in the training

```
algorithm SKIL(LEx,LAt,LAET,LFt) {
  // LEx, LAt, LFt  are lists of examples,
  // attributes and features. LAET  is the
  // list of attribute enabling triples.
  declare Node;
  NewLAt = OpenAttributes(LEx,LAt,LAET);
  Node.closed_ats =
        ClosedAttributes(LEx,LAt,LAET);
  if TestStop(NewLAt,LFt) {
    Node.type = (NewLAt==LAt?LEAF:H_LEAF);
    // H_LEAF, a concept hierarchy leaf.
    // LEAF, a tree leaf.
    return Node;
  }
  TransformFeatures(LEx,NewLAt,LFt);
  (At,TF) = SelectFeature(LEx,NewLAt,LFt);
  if (FtIrrelevance(LEx,At,TF)>MAX_IRL){
    // MAX_IRL - Max. feature irrelevance.
    Node.type = (NewLAt==LAt?LEAF:H_LEAF);
    return Node;
  }
  NewLFt = LFt - TF;
  for each TF_k in (TF.transformed_values)
   do {
    NewLEx = PartitionExamples(LEx,TF,TF_k);
    Node.sub_tree[k] =
        SKIL(NewLEx,NewLAt,LAET,NewLFt);
  }
  Node.type = (NewLAt ==LAt?TEST:H_NODE );
  // H_NODE, a concept hierarchy node.
  // TEST, a decision.
  return Node;
}
```

Fig. 4 — The SKIL Algorithm.

set is composed of a list of attribute-value pairs followed by a vector of feature values.

The algorithm (see Fig. 4) is a recursive procedure that takes as parameters a list of examples, a list of classification attributes, a list of attribute enabling triples and a list of features. In each stage of the induction, the main goal is to determine the values of as many classification attributes as possible. For each attribute, the discrimination power of features is evaluated, in terms of an entropy measure [7]. The feature that, for some attribute, gives the lowest entropy is selected to be test feature. Expansion stops when the values of all attributes have been determined or when it is not possible to extract more information from the data.

The basic knowledge transmutation used by SKIL is, therefore, empirical inductive generalization (see the Inferential Theory of Learning [5]), only that at multiple levels of abstraction. The generated knowledge structure is a hierarchy of anonymous concepts, each of them defined by the combination of several attribute-value pairs. The number of specified attributes and values defines the abstraction level. The formation of these concepts, guided by the attribute enabling triples, depends highly on the training data. The hierarchy is, simultaneously, a decision tree that can be used to recognize instances of the concepts.

3.2 Generating Behavior Taxonomies

Several experiments with SKIL were performed in the assembly domain [1,9]. A short summary is given here. Consider the macro-operation «Pick and Place» of a part and three of the basic primitives involved: a) approach to grasp position (`Approach-Grasp`); b) `Transfer` (of part); and c) approach to final position (`Approach-Ungrasp`). In the training phase, each of the selected operations was executed many times and several external exceptions were simulated. The trace of forces and torques in an interval surrounding each learning event was collected together with a description of the system behavior in terms of attributes as:

Attribute	Attribute Values
behavior	{ normal, failure }
part_status	{ ok, moved, lost }
failure_type	{ collision, obstruction }
collision_type	{ part, tool, front }

a) Attributes

Attribute	Attribute Value	Enabled Attributes
behavior	failure	{ failure_type }
failure_type	collision	{ collision_type }

b) Enabling triples

Fig. 5 — `Approach-Ungrasp` problem specification

behavior — generic information about the operation behavior; can be normal, collision, front collision or obstruction; what will be learned is, in fact, a model of the behavior (either normal or abnormal) of the system when performing these operations.

body — what was involved in the failure, e.g., the *part* , the *tool* , the *fingers* (*left* , *right* or *both* fingers).

region — region of body that was affected, e.g., *front* , *left* , *right* or *back* side , *bottom* , ...

object size — size of object causing failure: *small* , *large*.

object hardness — can be *soft* or *hard* .

object weight — can be *low* or *high* .

In this case, the goal of applying SKIL is to learn concept hierarchies characterizing the behavior of the system during execution of the selected operations under different external conditions.

For the operation `Approach-Ungrasp`, 117 classified examples were collected. The considered classification attributes and attribute enabling triples are presented in Fig. 5. The top-level (start) attributes are *behavior* and *part_status*. The discrimination features were extracted from raw force and torque data. After running SKIL on this domain specification and training data, a decision tree was obtained having 71 nodes. The concept hierarchy contained

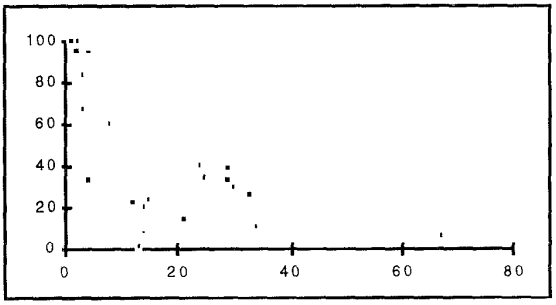

Fig. 6 — Error rates per attribute value versus the number of examples (`Approach-Grasp`).

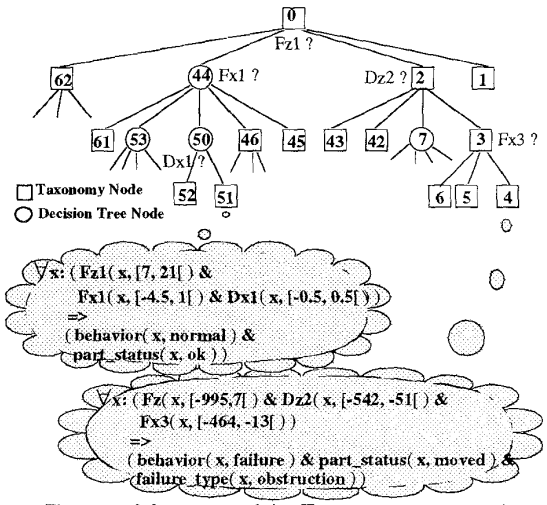

Fig. 7 — A fragment of the Taxonomy generated by SKIL for the `Approach-Ungrasp` primitive, and examples of the corresponding rules.

in the tree has 59 nodes, being 10 of them internal nodes and 49 terminal nodes (Fig. 7).

Performing the leave-one-out test with the same data and algorithm, the resulting average error rate is 15%. Experiments with traditional inductive learning algorithms, using the same training data and an equivalent set of classes (i.e., "flat" concepts corresponding to the combinations of the values of the classification attributes given to SKIL) produced error rates over 40%. Therefore, another advantage of SKIL seems to be to generate knowledge with a higher degree of accuracy.

When the user wants to get more and more information about a failure situation, the number of classification attributes and their values increases. If these attribute values are to be combined to produce "flat" classifications or labels, the number of labels increases exponentially, and the problem becomes intractable. This is the case of the information collected during execution of `Approach-Grasp`: 10 classification attributes, 28 attribute values and 8 enabling triples. Applying SKIL to this data (88 examples) produced a decision tree and concept hierarchy with 93 nodes. The resulting error rate (30%) is much higher than in the previous problem. This is understandable since the problem is much more complex and a smaller training set was provided. An equally complex specification was used in the `Transfer` problem, for which only 47 examples were collected. In this case the taxonomy generated by SKIL has an error rate of 34%. As could be expected from an inductive algorithm, when the number of occurrences of each attribute value in the training set increases, the corresponding error rate decreases (Fig. 6).

3.3 Failure Explanation

Training the system to understand the meaning of sensor values and learning a qualitative and hierarchical model of the behavior of each operation is a crucial step in diagnosis. Programming such model would be nearly impossible.

Since the human defines the "words" (attribute names and values) used in the model, the human is capable of understanding the more or less detailed description that the model provides for each situation. It is then easier to hand-code explanations for the situations described in the model. The explanation that must be obtained for the given execution failure includes, not only the ultimate cause (an external exception or system fault), but also the determination of the new state of the system. The failure explanation rules that we are using have the following format:

```
explanation(Op,FD,C,Del,Add) :- ... .
```

The first argument, `Op`, is the elementary operation during which the failure occurred or, in general, the execution context. The second argument, `FD`, is a description of the execution failure, obtained from the sensor data using the taxonomic model. Then, `C` is the ultimate cause of the failure. Knowing the exact cause can be irrelevant. What is important is to identify the state of the system after the failure. The arguments `Del` and `Add` specify the facts to be deleted from the state description and the facts to be added. For example:

```
explanation(
   get_tool(R,T1,TP1),
   [ [failure_type,wrong_tool],
     [current_tool,T2] ],
   unknown,
   [ current_tool(R,T1),
     tool_in_magazine(T2,TP2)],
   [current_tool(R,T2),Fact]
) :- ( toolplace(TPx),
       Fact = tool_in_magazine(T1,TPx);
       Fact = tool_lost(T1) ).
```

Diagnostic reasoning and causality have been studied for some time, and tested frequently in domains like electronic circuits, but there is no unified theory for these matters. Moreover, approaches like the one presented in [2] structure the problem considering that the main goal is to determine the faulty components in a system. However, in the assembly domain, not only system faults, but also external exceptions can be causes of off-nominal feedback. The described representation, that we are using, seems to provide useful results, although additional evaluation and refinement are needed.

4 Failure Recovery

Finally, when an explanation for an error is obtained, recovery planning is attempted. However, the most common case, probably, is that, not only one, but several possible explanations are found. The available sensor information is not enough to assert, with significant certainty, which is the state of the system. For instance, in the "wrong tool" example, presented above, the explanation rule provides several explanations, namely that the needed tool is in some other toolplace, TPx (and this is already several possibilities), or lost. In this case, the recovery

planner has to assume one of the explanations, generate a recovery plan and use it. If some problem arises, probably the assumption was not correct and some other explanation must be considered.

Note that the initial plan is generated for the nominal conditions. The representation of operators contains several assumptions. For instance, `not(defective(Obj))` is one of the pre-conditions of the `assemble` operator. Imagine that, in executing the assembly plan of the Cranfield Benchmark, mating one of the spacer pegs with the side plate fails. Various explanations are possible: the spacer peg might be defective, the side plate might be defective, some unexpected object originating in the environment might be obstructing the mate operation, etc. Unless some sophisticated sensorial feedback is available, the robot will have to recover based on assumptions or beliefs. The recovery actions will be, simultaneaously, verification actions.

One aspect that will be addressed in the near future is the application of learning techniques in error recovery. Some early planners (e.g. HACKER [13]) already included case-based reasoning features. These basic ideas have to be developed in order to take into account the complexity of real world problems. A learning feedback loop will have to be implemented [3]. In a first step, when an error, for which no recovery strategy is known, is detected, recovery planning is attempted. If recovery planning fails to generate a plan, eventually the human operator will provide one. In a second step, if the obtained recovery strategy is successful, it will be generalized and archived for future use. This will be attempted in the next phase of our work.

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. Research results concerning error detection and recovery in flexible assembly systems were presented. The proposed supervision architecture includes functions for dispatching of actions, execution monitoring and failure diagnosis and recovery. The research effort is directed towards something that really works. In exploring the scientific areas that may contribute to the development of the execution supervisor, we keep that goal in mind.

Developments in planning for nominal execution and for error recovery were presented. The used planning strategy is domain independent, non-linear, forward chaining, depth/best-first. The representation of the assembly domain knowledge was presented. Future research in this topic includes the learning aspects in recovery planning.

The lack of comprehensive monitoring and diagnosis knowledge in the assembly domain points out to the use of machine learning techniques, leading to an evolutive architecture. The general approach is to collect examples of normal and abnormal behavior of each operation or operation-type/operator and generate a behavior model that the diagnosis function will use to verify the existence of failures, to classify and explain them and to update the world model. Concerning the failure identification-classification problem, the application of the algorithm SKIL, that generates concept hierarchies with a higher degree of accuracy, provided interesting results. Further research will focus on efficient ways of generating examples, on feature construction and selection and on long-term learning.

Acknowledgements

This work has been funded in part by the European Community (Esprit project B-Learn and FlexSys) and JNICT (a Ph.D. scholarship and 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] de Kleer, J., A. K. Mackworth, R. Reiter (1990) Characterizing Diagnoses, *The Eighth National Conference on Artificial Intelligence (AAAI-90)*, Boston, p. 324-330.

[3] Evans, Ethan Z.; Lee, C. S. G. (1994) Automatic Generation of Error Recovery Knowledge Through Learned Reactivity, *Proc. of the 1994 IEEE Int'l Conf. on Robotics and Automation*, San Diego.

[4] Kaiser, M; Giordana, A.; Nuttin, M (1994) Integrated Acquisition, execution, evaluation and Tunning of Elementary Operations for Intelligent Robots, *Proc. of the IFAC Symp. on Artificial Intelligence in Real-Time Control*, Valencia, Spain.

[5] Michalsky, R.S. (1994) Inferential Theory of Learning: Developing Foundations for Multistrate-gy Learning, *Machine Learning. A Multistrategy Approach*, vol. IV, Michalsky, R.S.; Tecuci, G (eds.), Morgan Kaufmann Pub., San Mateo, CA.

[6] Mozetic, I. (1991) Hierarchical Model-based Diagnosis, *Int. J. of Man-Machine Studies*, 35, pp. 329-362.

[7] Quinlan, J. R. (1986) Induction of Decision Trees, *Machine Learning* , 1, pp. 81-106.

[8] Reich, Y. (1994) Macro and Micro Perspectives of Multistrategy Learning, *Machine Learning. A Multistrategy Approach*, vol. IV, Michalsky, R.S.; Tecuci, G (eds.), Morgan Kaufmann Pub., pp. 379-401.

[9] 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* (to appear), Nagoya, JP.

[10] Seabra Lopes, L. Camarinha-Matos, L.M. (1995) Planning and Supervision in Assembly Systems: a Machine Learning Approach, *Proc. 3rd European Workshop on Learning Robots*, MLnet Familiarization Workshop Series, Heraklion, Crete, Greece.

[11] Seabra Lopes, L.; Camarinha-Matos, L.M (1995) Planning, Training and Learning in Supervision of Flexible Assembly Systems, *Balanced Automation Systems: Proceedings of BASYS'95* (to appear), Vitoria, Brazil.

[12] Sacerdoti, E. (1977) *A Structure for Plans and Behavior*, Elsevier-North Holland.

[13] Sussman, G. (1975) *A Computer Model of Skill Acquisition*, Elsevier, New York.