Section II

Integration and Learning in Supervision of Flexible Assembly Systems

Luis M. Camarinha-Matos, *Member, IEEE*, Luis Seabra Lopes, *Student Member, IEEE*, and José Barata, *Member, IEEE*

Abstract— **A generic architecture for evolutive supervision of robotized assembly tasks, in a context of integrated manufacturing systems, is presented. This architecture provides, at different levels of abstraction, functions for dispatching actions, monitoring their execution, and diagnosing and recovering from failures. The problem of integration of legacy systems is discussed and an implementation approach described. Modeling execution failures through taxonomies and causal relations plays a central role in diagnosis and recovery. 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. Methodologies used, performed experiments, and obtained results are described in detail.**

I. Introduction

THE NEED for flexible assembly supported by intelligent supervision systems, largely anticipated by the research community, is becoming an actual competitive factor for industrial enterprises. The increasing globalization of the economy, following the openness of markets and the tendency to the creation of economic blocks is imposing tough challenges to manufacturing companies, leading to the concept of lean/agile manufacturing. Among other vectors, this situation stresses the need for truly flexible manufacturing and assembly systems (FMS/FAS).

The development of a flexible assembly system (FAS) imposes important requirements both from the manufacturing equipment and from the software architecture points of view. In terms of hardware, examples of such requirements are: the use of multioperation devices and robots with multiple tools/end-effectors, modular design of fixtures and feeders, rich sensorial environment, advanced communication infrastructures and standard protocols (Manufacturing Automation Protocol/Manufacturing Message Specification (MAP/MMS) [21], [23], FIELDBUS [17], TCP/IP), flexible cell organization (logical cells), etc. Related to software, resorting to "intelligent" functionalities arises as a natural requirement.

Manuscript received November 23, 1994; revised December 28, 1995. This work was supported in part by the European Community (ESPRIT Project B-LEARN and ECLA FlexSys) and JNICT, the Portuguese research board (Project SARPIC, Project CIM-CASE, and a Ph.D. scholarship).

The authors are with the Departamento de Engenharia Electrotécnica, Universidade Nova de Lisboa, 2825 Monte da Caparica, Portugal.

Publisher Item Identifier S 1042-296X(96)02538-4.

1) Interactive programming/planning, in order to reduce set up costs and to assure a rapid response to product innovation, process changes, or shifts in demand.
2) Sensorial perception and status identification.
3) On-line decision capabilities, such as monitoring, diagnosis and error recovery, in order to cope with the nondeterministic behavior of less structured cells.
4) Information integration, in order to consider the FAS system as part of a more general computer integrated manufacturing (CIM) system.

This requires a comprehensive world model and the capability to update such model, in run time, with perceived information coming from sensorial systems and from the human interlocutors. Therefore, an important requirement for a FAS will be an interactive planning and intelligent supervision system.

Execution supervision of industrial assembly tasks must be understood in the general framework of a CIM environment. This activity depends on and interacts with various other activities involving specialized knowledge. The complexity and wide range of such areas, involving different expertises, make non realistic some "isolated" approaches to the assembly planning and execution supervision. The "game" has to be understood in the context of interaction between areas that are evolving in parallel, trying to benefit from their results and influence them in a way leading to a convergent global architecture.

On the other hand, this approach is consistent with the tendency reflected by concurrent engineering [7], [29]. The concept of concurrent engineering has become more and more popular in recent years as a result of the recognition of the need to integrate diversified expertise and to improve the flow of information among all "areas" involved in the product life cycle. Evolving from earlier integration attempts, represented by the paradigms of "Design for Assembly"/"Design for Manufacturing," concurrent engineering is a consequence of the recognition that a product is the result of many factors, including marketing and sales factors, design factors, production factors, usage factors (intended functionalities/requirements), and destruction/recycling factors. The principle of concurrent engineering is therefore to create a team of experts from different fields and make the team responsible for the design,

engineering, manufacturing, and marketing of a single product. For instance, production engineers in the team must be able to look at the early designs, just as well as sales people. This is radically different from the traditional approach known as finish the design and "throw it over the wall" to the production engineers, and hardly interact with them at all. Team work, based on concurrent or simultaneous activities, potentially leads to a substantial reduction in the design-production cycle time, if compared to the traditional sequential "throw it over the wall" approach, as well as to an improvement of quality and cost reduction.

The need to understand the interactions between planning, scheduling, and execution supervision is just a part of this general concept. To design a supervision architecture, one must be aware of what (information) can be expected from the levels above. Aiming at a new generation of intelligent supervisors, it may be necessary to ask additional information, not required by traditional controllers, from the preceding activities. On the other hand, from the operational perspective, problems not solved by the supervisor have to be fed backward to preceding planning activities. For instance, an unrecoverable error may imply a rescheduling (dynamic scheduling [31]) or even a replanning. Knowing the structure and information manipulated by these activities may tune the new demands in order to be realistic.

Planning in manufacturing and assembly is typically done in a hierarchical fashion [6], [10]–[12]. At a more abstract level, processes related to long-term planning, such as strategic planning, marketing planning, production planning and high-level performance monitoring, will be carried on. At an intermediate level, operational planning processes, like master production scheduling, materials requirements planning and capacity planning are considered. At the next level, product and process oriented planning, and scheduling, will be placed. The traditional assembly task planning can be decomposed in two main phases: product oriented planning and process oriented planning. The first phase, based on the product model (bill of materials, geometric model, tolerances model, materials model, etc.), determines a feasible assembly precedence graph taking into account the constraints derived from the product model itself. The second phase generates, from this precedence graph, an executable assembly plan, taking into account the cell structure and functionalities and technologic knowledge. This phase is typically split in two steps: process planning and detailed execution planning, from which intermediate plan representations can be produced.

The process plan is still a high level representation of the assembly task, based on abstract operators, like `Peg-into-hole`, `Transfer`, `Insert-screw`, `Position`, etc., but already taking into account feasibility conditions from the technologic point of view. Various interactive planning systems—e.g., computer aided process planning (CAPP)—have been developed in recent years [16], [43] to help the human expert in generating appropriate process plans. Most of these systems follow an interactive generative approach, in the line of a decision support system, helping the production engineer in the generation of a feasible sequence of assembly steps. Each node of the process plan typically includes information such as:

- abstract operator to be applied;
- operands, i.e., parts to be assembled;
- mating referentials (goal positions), or approach directions;
- suggested part grasping zones;
- selected resources type (robot and end-effectors);
- operation duration.

When a specific assembly cell is decided/selected, the task specification can be further refined, through several steps, until a description understandable (i.e., executable) by the cell controller is reached.

Several attempts to adapt generic planners, developed in the AI community, to realistic robotic tasks have been made. Most of these planners were designed having the "blocks world" in mind and some of the approaches have just tried experiments in very particular situations that could be accommodated to a simplified world model. In some of these works, a link to a real robot was established, the elementary operators, generated by the planner, were translated into the real syntax of the robot control language, but demonstrated tasks are limited to the manipulation of very simple objects ("blocks"), under strictly constrained movements (grossly discretized space). Some applications like pick-and-place or palletizing, can be realized with such a simple approach. For more complex tasks, as in assembly, additional capabilities are needed especially in what concerns spatial reasoning/planing of flexible motion. Other approaches are strongly geometric-reasoning-based [1], [20], [32] but require heavy processing procedures and achieved results still present some limitations. On the other hand, the most adequate spatial-related solutions are not completely justified by pure geometric reasoning but depend on other technological constraints. Therefore, we do not consider totally automatic planning approaches as realistic.

Alternatively, taking into account the multistage planning process described above, a less automated but more realistic approach to assembly planning can be pursued. The interactive approach we followed in some experiments [6], [10] assumes the availability of a "rich" task specification resulting from product design and process planning phases. The description resulting from these phases is refined by normal hierarchical planning techniques and resorting to graphical simulation to acquire, from the human expert, positioning information (grasping, approaching, trajectory skeletons, etc.). In this way, graphical simulation is, not only a mean to verify/evaluate a generated plan, but also an instrument to help the interactive construction of such plan. It should be noted that a similar approach has been successfully applied to other domains (e.g., welding tasks [41]). It seems reasonable to apply the same strategy on assembly tasks that, in fact, are of a greater complexity.

Typically various manufacturing orders may be competing to be executed on the same production resources. Alternative plans may also be available for the same product, depending on the process and the resources to be applied. Therefore, a scheduling activity takes the various executable plans, as well as other information, like the job size, the order due date,

the available manufacturing resources, and some optimization criteria, and produces a refined executable plan which includes, in association with each operation, the assigned resource and time window.

The work reported bellow is part of the development of an execution supervisor, which receives as input an executable assembly plan generated as described above and will carry out its execution, performing monitoring, diagnosis and recovery functions. The executable plan produced according to the mentioned multistep procedure is assumed to have a hierarchical structure, which leads to more modular supervision activities [11]. On the other hand, since the planning activity is often carried out hierarchically, the generation of the proposed plan structure requires no additional effort. From the supervision point of view, the hierarchical approach can be combined with concurrent execution at each level. In the lowest level, primitive resource operators, like Move or Grasp, are considered. Fine motion and compliant actions, like Peg-into-Hole, are considered primitive operators. At the next level, immediately above, operators like Pick or Mate, are included. In the upper levels of the plan, the operations are process-dependent and represent important logical phases of the plan execution. At each level, plan operators are modeled in STRIPS (STanford Research Institute Problem Solver) style [38].

The work being conducted in our labs considers, at different levels of abstraction, functions for dispatching actions, monitoring their execution, and diagnosing and recovering from failures. An important aspect in this context is the evolution from legacy systems. Any realistic approach to more advanced manufacturing and assembly systems has to take into account existing systems and components and find an appropriate transitional procedure. As an example, the "opening" and partial "reconfiguration" of existing device controllers, in order to integrate them into a cooperating community, is not a negligible task, as most of these controllers were designed under a stand alone perspective. Another main problem is the acquisition of knowledge about the environment in order to support monitoring, diagnosis and recovery. For this purpose, the use of machine learning techniques is being investigated. The integration and enhancement of existing controllers and sensorial subsystems as well as preprocessing techniques for sensorial data (signals to symbols conversion) is a major requirement in order to integrate symbolic machine learning techniques with real robotic systems. Results achieved under this approach in the context of the European ESPRIT project B-LEARN II are described as well as the planned extensions and main foreseen difficulties.

II. Supervision System

As already suggested, the architecture of an intelligent execution supervisor should reflect the hierarchical structure of the plans. For each plan level, its main functions are [9] and [11].

1) Dispatching and Global Coordination: The global coordination activities performed by a high level controller include the following: dispatching actions to the executing agents, driven by the scheduled task plan; synchronization of activities

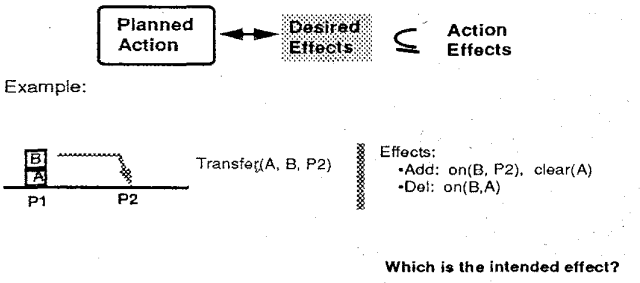


Fig. 1. Action effects and action goals.

performed by different agents and synchronization with external events; and world model update and information exchange, resorting to a cell information system or knowledge base. The dispatcher also coordinates the execution of the other modules of the intelligent supervisor as well as the interaction with the human operator.

2) Monitoring of Assembly Plans: The monitoring function is used to detect nonnominal feedback in the system during the execution of assembly plans. Two monitoring modes are usually considered: discrete monitoring and continuous monitoring [24]. Discrete monitoring checks preconditions before the execution and goal achievement after the execution of operations. Continuous monitoring checks sensory conditions during the execution of operations.

For reasons of efficiency, the supervision system should only monitor the achievement of the intended effect (Fig. 1). That is, during planning phases, the intentions of the planner when selecting an operator should be "stored" together in the corresponding operator node in the plan.

In many contributions to the problem of execution monitoring, the sensory conditions to test in each situation are defined in a model [24], often coded as monitoring rules [8], [22]:

$$\text{IF } \langle \text{situation} \rangle \text{ AND } \langle \text{sensory-condition} \rangle \text{ THEN } \langle \text{actions} \rangle.$$

However, very often it is not easy, even for an expert, to specify the sensory conditions that guarantee the success or the preconditions of an action and to identify the statistical significance of each situation. The use of machine learning techniques may help to relate the conditions specified in a plan to testable sensors.

3) Failure Diagnosis: The diagnosis function will firstly check if there really is a failure (failure confirmation) and update the internal model. Then, this function will try to classify and explain the failure. In the early work of Srinivas [42], for each action, a failure reason model is built, which specifies the collection of all possible failures and all features that are expected to manifest for each failure. At each execution level, different levels of explanation for a detected failure may be generated, depending on the amount of information available [9]. For example, a gross diagnostic can be "pick fail." A more detailed diagnostic could be "pick fail due to object sliding." The least detailed explanation would be "deviation detected."

In [35] it was proposed to divide errors in three main families: system faults, external exceptions and execution failures (Fig. 2). **Execution failures** are deviations of the state of the

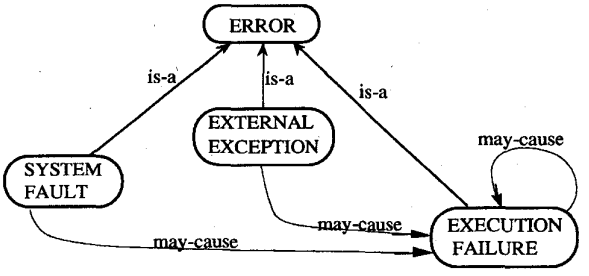


Fig. 2. Typical relations between the main causes of errors.

world from the expected state detected during the execution of actions. For example, collision, obstruction, part slippage from the gripper, part missing at some expected location, etc., are execution failures. **External exceptions** are abnormal occurrences in the cell environment that may cause execution failures. For instance, misplaced parts, defective parts, and unexpected objects obstructing robot operations may cause execution failures. **System faults** are abnormal occurrences in the hardware and software of assembly resources and in communications. Generally speaking, these are not errors that the system can recover from, unless some functional redundancy is available and rescheduling is performed. However, the system must be prepared to detect and identify this type of errors in order to prevent the occurrence of other errors.

4) Failure Recovery: At each supervision level, the recovery function is called when the diagnosis function confirms a failure and finds an explanation. The recovery function will try to determine a recovery strategy to bring the execution to a nominal state. One basic question is how to build recovery strategies? Since the detected error is some unexpected (abnormal) event, the nominal plan is not to be altered.

In the proposed hierarchical approach, when a failure is detected before, during, or after the execution of an action, and it is not possible to classify, explain, or recover from that failure, the problem is passed on to the next upper level where context information is broader. In the lower levels of the plan, recovery actions will tend to be simple reflexive actions, while in the upper levels determining recovery actions will require more extensive diagnosis and planning.

In Fig. 3, an example of the whole error detection and recovery cycle is presented. 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. In this example we consider that, during the execution of a Transfer operation, in which the robot carries a part to be assembled, an object, unexpectedly appearing in the environment, collides with the gripper causing the part to move without falling. The first diagram, included in Fig. 3, shows the perceived sensor data during actual execution. 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 operator model. Since a deviation is detected, the diagnosis function is called to verify if an execution failure occurred and, in that case, determine a failure classification and explanation. For this function, additional features must

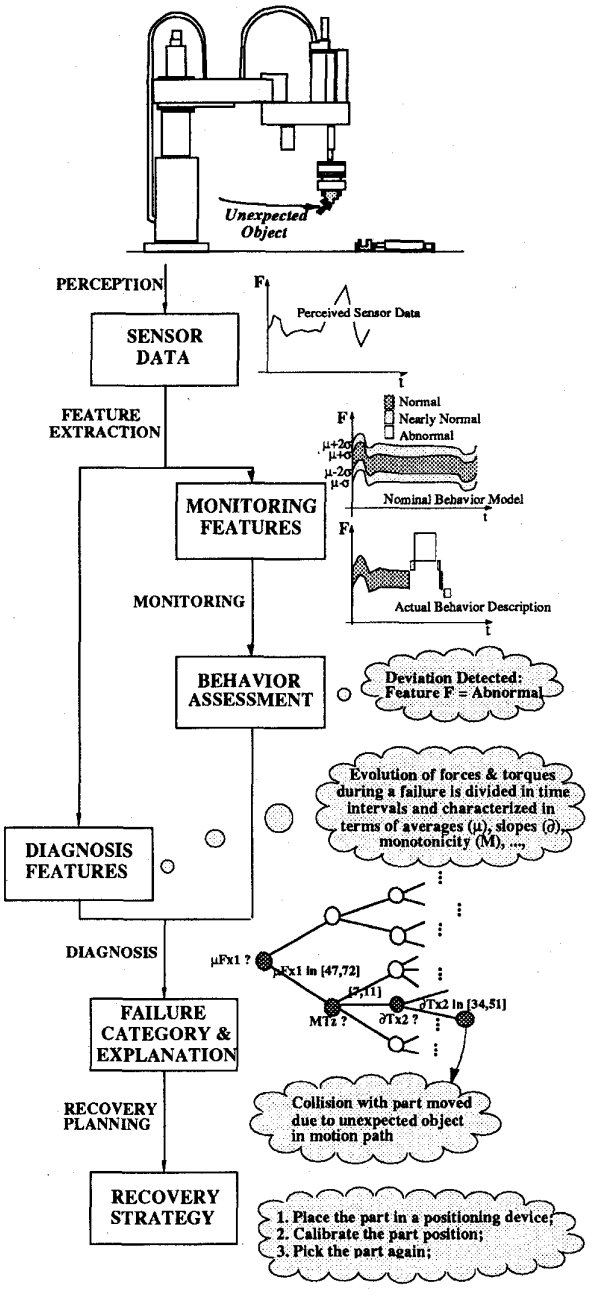


Fig. 3. The error detection and recovery cycle—Example.

be extracted, as it will be explained in Section IV. Diagnosis is a decision procedure that needs a model of the task, the system and the environment. The final step, based on the failure characterization, is recovery planning. In the example, the robot should place the part in a positioning device and regrasp it after position calibration.

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 explana-

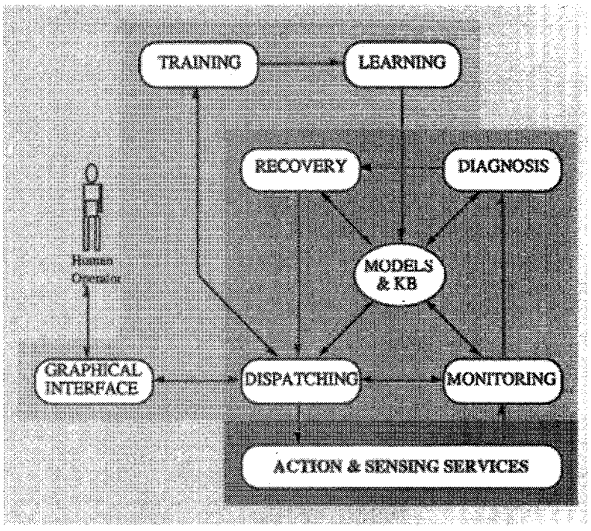


Fig. 4. An architecture for autonomous supervision.

tions 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. 4). 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. In Section IV, a detailed description of this learning functionality is presented.

III. Cell Modeling and Integration

In order to install an Intelligent Supervisor on a FAS, an execution infrastructure, providing integrated access to the local controllers of the manufacturing resources, is necessary. The increasing demand for highly sophisticated supervisors implies local controllers with sophisticated features. Although existing controllers are not suited to provide this kind of requirements, it would not be realistic to ignore them and start everything from scratch. To overcome this, it would be necessary to "adapt" existing controllers (legacy systems) to the new reality by developing an abstract machine that hides the hardware peculiarities and provides new sophisticated services to its client (the supervisor).

A. Migration from Legacy Systems and Cell Integration

Migration from and/or integration of legacy systems is one of the most challenging aspects of manufacturing systems. The existing gap between functionalities provided by currently available controllers and functionalities needed by an intelligent supervisor requires a considerable effort to be overcome. These controllers were developed with completely different purposes, to be used in stand-alone operation, almost without functionalities to cooperate with other components. Components were developed to be used as "masters" of a "small kingdom" and not as agents to be controlled by high level controllers/supervisors.

A migration procedure is necessary to "recover" existing controllers. Creating entirely new controllers, with functionalities adapted to the supervisor requirements could be a form to overcome that problem, but it would imply a tremendous cost. Existing controllers, by economical reasons, cannot be simply thrown away. Therefore, a more sensible approach is to try to adapt existing systems to the requirements of high level controllers, which need only a smaller set of functionalities but a larger openness.

The requirement of the high level controller to directly command legacy systems operations is one of the most important aspects. Most of the existing controllers provide no way to do that. To fulfill this requirement an interpreter or adapter in the controller side should be developed to accept commands that could be issued via an input/output port. It should be pointed out that, in most cases, this interpreter will reduce the functionalities of the local controller, but the gain coming from the possibility to have a controller that can be easily integrated in a manufacturing system overcomes these disadvantages. Developing the interpreter is not an easy task, since robot manufacturers do not have the tradition to develop open architectures. To add anything to the system, other than developing programs using the manufacturer's own development tools, is a hard job, requiring a tremendous effort in "breaking" protocols and adapting controllers functionality to the new requirements.

B. Cell Modeling

In our approach, to connect an intelligent supervisor to real components it is necessary to build up a software layer that provides the functionalities needed by the supervisor. This software layer can be seen as an abstract machine that supplies services to a high level supervisor in the same sense an operating system provides services to applications. The clear separation between the supervisor and the abstract machine allows for a transparent access, hiding the hardware peculiarities and the heterogeneity of the various local controllers from the supervisor (Fig. 5). In this sense, the services exported by the abstract machine should be independent from the specific hardware, meaning that the variety of existing controllers should be integrated by this platform. The abstract machine services should not be the sum of all individual features that exist in each real controller. This is justifiable because: 1) available controllers show a big heterogeneity in terms of the level of abstraction they export; 2) the behavior of a particular controller is constrained, i.e., has to be coordinated according to its role in the integrated community of agents. Therefore, services offered by the integrating platform should be of higher level, resorting to the low level services (existing in the real controllers), but they include some knowledge about the

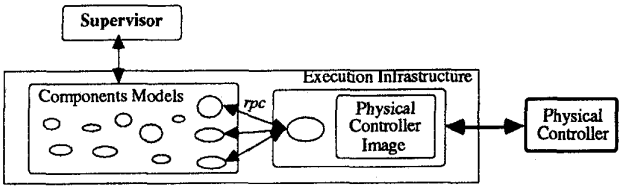


Fig. 5. Integration of local controllers in an open environment.

structural aspects of the underlying system. A rich underlying system model (abstract machine) will simplify the supervision activities.

In order to develop such abstract machine using the proposed requirements, it is necessary to choose an adequate modeling paradigm. Two modeling perspectives can be pointed out clearly: modeling of cell/system structural aspects and modeling of the static and dynamic properties of individual components. The combination of object-oriented and frame-based programming paradigms seems to be suitable for this purpose, due essentially to their constructs to model the operational aspects of the components (methods/demons) and to represent the structural aspects of the system (relations that can be user defined). Methods and demons associated to the component's model can hide the underlying hardware infrastructure. Another important aspect is the "relation" concept, which can provide a flexible way to describe inter-components' relationships. The Golog frame engine, developed in-house, is being used [34].

1) Structural Model: In the following discussion, the basic modeling unit will be a cell. A cell is a composite entity that is capable of performing some transformation, movement or storage related to some product or part [3]. In structural terms, each cell has components to support the input of parts, an agent to perform the transforming actions and components to support the output of products/processed parts. An example of a cell model can be found in Fig. 6. The relation *connected_from* links the cell model to the entity or entities performing input activities. The relation *connected_to* links the cell model to the entity or entities performing output activities. The relation *processor* links the cell model to the agent performing transformation activities. The generic cell concept can be specialized by activity. There can be cells specialized in assembly, painting, welding, storage, machining, transportation, etc. A shop floor is just a set of specialized cells. The input and output activities can be performed by several agents, i.e., there may exist several candidates, depending on the application.

At this stage it is convenient to make a distinction between the concepts of agent and component or manufacturing resource. For instance, the model of a robot **component** is a context independent description of its static and dynamic characteristics. A robot **agent** is a model of a robot and associated resources, like tools or auxiliary sensors, when inserted into a particular context. A robot can play different **roles** in different **contexts.** The (expected) behavior of a robot in an assembly context is different from its behavior in a spot welding context. On the other hand, when a robot is performing a given role, it resorts to auxiliary resources,

```
Frame: cell {
  base_coordination_system:
  processable_products:
  input_parts:
  connected_from :
  processor :
  connected_to :
}
Frame: Assembly-Cell {
  is-a : cell
  val-inp-ag: buffer, gravit_feeder,
      index_table, agv, conveyor
  val-out-ag: conveyor, agv, buffer,
      index_table
  val-proc-ag: robot
```

Fig. 6. Cell and assembly cell.

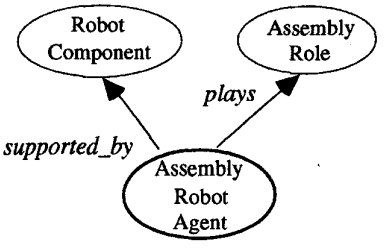


Fig. 7. Structure of an assembly robot agent.

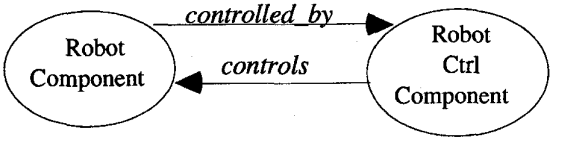


Fig. 8. The relations *controlled_by* and *controls.*

like tools, sensors, buffers, etc., that extend its functionality in order to fulfill the functionality required by the role. A robot agent is, therefore, a model of the robot when playing a particular role and extended by selected attributes inherited from the auxiliary resources (Fig. 7).

The relationship between a robot component and its controller can be found in Fig. 8. The relation `controlled_by` links the model of the robot to the model of its controller (Figs. 8 and 10). The operations specified in the inherits slot are inherited by the robot component. In assembly applications, a robot could have the role exemplified in Fig. 9.

In a role, *main_attributes* is a slot related to the inheritance mechanism of the *plays* relation. In this case, it specifies which are the characteristics of the assembly role that will be relevant to a processor agent. The slot `component_attributes` has the same functionality as *main_attributes,* but, in this case, associated to the relation *supported_by.* This slot describes the most relevant component attributes that are important to the processor agent. The slots `tools_domain` and `aux_res_domain` represent domain knowledge that is important. The relation *current_tool* associates the main player of this role (robot component) to a particular tool. The relation

```
Frame: assembly_robot {
  is-a : agent
  plays : assembly_role
  supported_by : robot_component
}
Frame: assembly_role {
  is-a : role
  tools_domain: (grippers, screwdriver)
  aux_res_domain: (buffers)
  force_sensor :
  current_tool :
  available_tools: gr1, gr2, sd2
  assembly_device : fixture1
  main_attributes: force_sensor,
          current_tool, ...
  component_attributes:
  base_coordinate_system,
      controlled_by, current_position
}
Relation: plays {
  is-a : relation
  type: intransitive
  inherits: inclusion(main_attributes)
  inverse_relation: played_by
}
Relation: supported_by {
  is-a : relation
  type: intransitive
  inherits : inclusion(component_attributes)
  inverse_relation: performs
```

Fig. 9. The robot agent.

assembly_device specifies where assembly operations are really done (e.g., in Fixture1).

Finally, an example of an agent is described in Fig. 9. The relation *plays* associates an agent with a specific role. The relation *supported_by* associates an agent with its intrinsic properties (component). The inheritance slot specifies the slots to be inherited by the processor agent.

2) Dynamic Model: The dynamic model is related to the way components' physical changes are reflected in the model and vice-versa. Components' physical behavior is realized by controllers actions, i.e., robot movement or part feeding operations are actuated by a controller. Internal component models should reflect the physical behavior. In this way, it is natural to consider that models should have the same kind of operation, i.e., behavior is described by an entity that virtualizes the functionality of the physical controller. Every component has a *controlled_by* relation to assign a controller model to a component. This is the way component models represent behavior. For example, an instance of **robot_component** should be related to an instance of **robot_ctrl_component** via the *controlled_by* relation (Fig. 10). This frame defines all methods/demons that virtualize the physical controller's functionality. Connection between methods or demons and the physical controller is made through a server that is a kind of physical controller's "mirror." Using a method or a demon depends on the existence of a variable associated to its corresponding behavior. For instance, the behavior of feeding a part is described by a method, while a robot movement

```
Frame: robot_component {
  is-a : manufacturing_component
  base_coordinate_system:
  controlled_by :
  applications: assembly, gluing, ..
  current_position:
}
Frame: robot_ctrl_component {
  is-a : controller
  move_wc: method move_wc_fn(x, y, z, q)
  move_jc: method move_jc_fn(m1,m2,m3,m4)
  hardhome: method hardhome_fn
  acceleration: demon if_write accel_dem
  speed: demon if_write speed_dem input
  output: byte demon if_write output_dem
}
Relation: controlled_by {
  is-a : relation
  type: intransitive
  inherits: inclusion(move_wc,move_jc,...)
  inverse_relation: controls
```

Fig. 10. Robot and its controller.

could be described by a demon associated to the attribute that represents robot's position. This means the existence of variables that describe the controller's internal state, which will be accessed through their attached demons.

During physical controller's operation, the model can provide updated values of control variables when consulted. From the controller's model point of view, these variables are **persistent,** because they "keep existence" over its execution. No concern is necessary to save these variables between supervisor's executions, as they persist in the physical controller. This persistence does not imply a static behavior because there exists a dynamic link, implemented by a demon, between the supervisor side and the physical controller side. Any change in one side implies the other side's awareness and therefore a **dynamic persistence** concept is achieved. Reactive programming is useful to implement the link between the model and the dynamic aspects of the components (Fig. 11).

Control variables actuated externally (digital inputs) are associated to *if_read* demons. Every time a client (supervisor) of the controller server needs to know the value of an external input, it performs a read operation of an attribute that reflects the state of that input. This read action fires the demon which accesses the external controller to read the input value.

On the other hand, control variables actuated internally (robot position) are associated to *if_write* demons. Every time a client (supervisor) needs to change the robot's position, it just changes that attribute, implicitly firing an *if_write* demon, which sends the necessary commands to move the physical robot to the new position. But if the client only needs to know the current position, it just reads the corresponding attribute, firing an *if_read* demon, which sends the commands necessary to get the position from the physical controller (Fig. 11).

With this approach, an abstract representation of a cell can be developed to be used in the supervisor architecture. This representation allows for physical control actions as well as sensorial feedback to the supervisor.

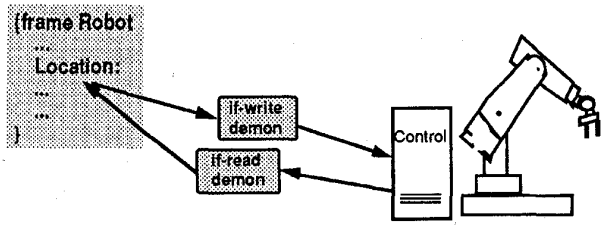


Fig. 11. Use of reactive programming to support dynamic persistence.

IV. Reasoning About Failures—A Machine Learning Approach

As emphasized in Section II, the difficulty in hand-coding the models that the monitoring, diagnosis and recovery functions need, raises the question of how to build such models automatically. The use of machine learning techniques seems a promising approach to the problem. In the following, some methodological and experimental developments in applying inductive learning to generate diagnostic knowledge are presented. Particular attention is also given to the training methodology as well as to obtaining qualitative representations of normal and abnormal system behavior.

A. A Qualitative Reasoning Perspective

In some early approaches to error recovery in robot programs, it was already understood that the use of good domain knowledge was important but it should be combined with some sort of common-sense and qualitative reasoning [18]. For instance, it is very difficult to model friction mathematically. Still, humans, using their "fuzzy" understanding of the physical world, can deal with friction in every-day tasks. There is already an important body of literature coming from the area of qualitative physics/reasoning [15]. However, being quite interesting the available results, they are at the same time a little disappointing, since codifying qualitative knowledge about the physical world turned out harder than expected.

The fact that many of the test cases analyzed by researchers in this area are in the domain of continuous processes (for instance in chemical plants) seems to have taken them to believe that things change smoothly in the physical world. However, in the robot assembly domain, this assumption does not hold. Errors occur unexpectedly, causing system parameters to change abruptly. Furthermore, the overall nominal execution of an assembly plan cannot be considered a continuous process. At most, some of the primitive actions in the plan can be considered to have a certain degree of continuity. And yet, the execution of an assembly plan is certainly a physical process that, at a certain level of abstraction, should be possible to describe qualitatively. In the following we describe ways to obtain qualitative representations of numerical sensor data relevant to monitoring and diagnosis, then give an overview of our current ideas about how the model of errors should look like and how to use it, and finally present the learning techniques used.

1) From Signals to Symbols: In robotics and automation environments, the richer sources of information about the status of the system are, often, sensors that return numerical data difficult to analyze. The main source of numerical data in our experimental setup is the force and torque (F&T) sensor. Data coming from other sensors, most of them binary, can be directly mapped to information on status of feeders, fixtures and tools. With the F&T sensor we can monitor actions in which the robot arm is involved. It would be desirable that the execution supervisor could reason about the evolution of force and torque values, measured during the execution of actions, in terms of its overall characteristics, and not in terms of the individual numerical values, i.e., in short, as humans do.

In the field of qualitative physics, the frequently proposed representations for numbers include signs, inequalities, and orders of magnitude. Fuzzy logic could be used to model qualitative values of numerical variables. For instance, consider the behavior of a sensor variable during a certain period of time. A human, making a qualitative description of such behavior, would probably divide it into intervals, and would mention roughly how long these intervals were, which were the average values in each interval, as well as the average derivatives. Thus, fuzzy descriptions for time intervals, amplitudes and derivatives are needed. These descriptions can be given or learned.

Dealing with time intervals is not an easy task, mainly when the goal is to apply existing machine learning algorithms to generate new knowledge. Currently, we divide numerical sensor data behavior traces in a fixed number and equal length set of subintervals. For each of them, averages, slopes, etc., are calculated. The generic approach is to calculate, from the raw sensor data, features closer to the way humans think. The second step is the derivation of a symbolic description of these higher level features.

Since one of the goals is to generate classification knowledge about execution failures, based on provided examples, a method was developed [35], [36] to generate symbolic descriptions of numerical features which maximize their class discrimination power. This method follows three steps. The first step is to produce histograms for all pairs of classes and features. Each histogram shows the number of examples of each class corresponding to several intervals of values of the feature. The number of intervals considered is given by the number of Struges: $\text{Interv} = 1 + \log_2 \text{Tot}$, in which Tot is the total number of examples represented in the diagram, i.e., the total number of examples belonging to that class. In the second step, each histogram will be approximated to several well known statistical distributions (e.g., normal, exponential, uniform). The chi-square test will determine which distribution fits better in the histogram. To apply this test, the relative frequency of objects of the considered class in each interval i, qi, is calculated according to the distribution being tested. Let N be the total number of objects of the class in the training set, and N_i the number of objects of the class in interval i. χ^2 is defined as

$$\chi^2 = \sum_i \frac{(N_i - N \cdot q_i)^2}{N \cdot q_i}.$$

The distribution that gives the lowest value of χ^2 will be chosen. The last step is to apply a rate of significance to

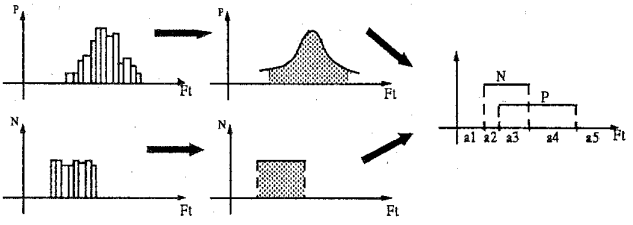


Fig. 12. Numerical to symbolic conversion.

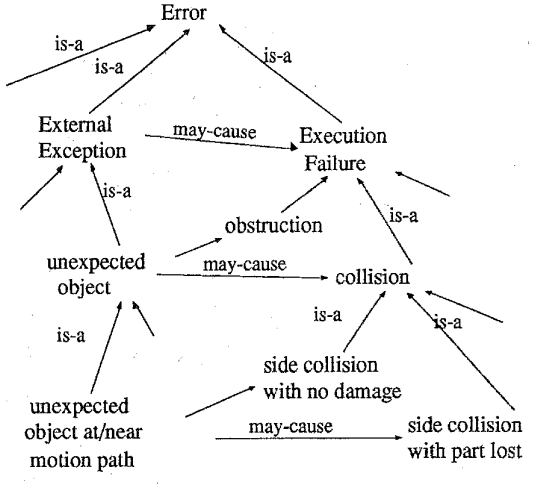


Fig. 13. Example of causal links at different levels of the error taxonomy.

the distribution in order to ignore values of the feature that do not occur significantly in the examples of the considered class. Finally, the intersection of the intervals of values of the feature, in which each class may occur, defines qualitative values for that feature (Fig. 12).

2) The Model of Errors: Depending on the available sensorial information, a more or less detailed classification and explanation for a detected execution failure may be obtained. Therefore, the model of errors should be a taxonomy. At each level of this 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. Determining explanations for detected execution failures can become very complex when errors propagate. The proposed approach to modeling errors in terms of taxonomic and causal links aims at handling this complexity (Fig. 13) [9], [28], [35].

In actual execution, when a failure is detected, the current state of the world is analyzed, as well as its evolution in an interval surrounding the time of detection, and a failure classification is determined. This can be done using knowledge generated by inductive machine learning techniques, as it will be described bellow. Then, the model of errors will be used to explain the failure, i.e., to determine its causes. 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 [14] 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.

B. Learning: Previous and Related Work

In previous work, concerning the diagnosis functionality [11], [35], [36], an inductive learning algorithm [19] was applied. The algorithm is simple and, compared to ID3 [30], AQ [25] and other well known symbolic inductive learning algorithms, has the advantage of dealing elegantly with continuous training data. The disadvantage is that it does not consider discrete features.

The learning kit CONDIS (inductive learning in continuous and discrete domains), developed in-house [37], may be used in domains characterized by symbolic and/or numerical features. It was designed to be easily integrated in an application. It can be used to test different approaches to a particular learning problem. The objects in a domain are characterized by a set of attributes or features and can be grouped according to a set of classes. Continuous features take numerical values while a discrete feature takes one of a finite prespecified set of values. The calculation cost for each attribute may also be provided.

As pointed out by Cheng *et al.* [13], a symbolic inductive learning algorithm (decision tree generator) is a recursive procedure for which four rules must be specified.

- R1. *Test Stop:* a rule for deciding when to stop the recursion, i.e., when to create a leaf node.
- R2. *Classify:* a rule for labeling a leaf node with a class.
- R3. *Select Feature:* a rule for selecting a test feature.
- R4. *Partition Examples:* a rule for partitioning a set of examples.

The CONDIS learning kit, implemented in C for efficiency reasons, allows the user to define such rules. In addition, for domains characterized by complex features, it was included in CONDIS the possibility of defining rules for transforming, in each step of the induction, the set of values that the features can take. Considering a rule for the initialization of the induction process (e.g., for data structure initialization or for generation of a new table of examples in terms of higher level features) might also be useful.

- R5. *Transform Features:* transformation of the domain of values of a feature.
- R6. *Init Induction:* initialization of the induction process.

This system is being used to evaluate different variations of the classical structure of inductive algorithms. For example, if the appropriate rule implementations are provided, CONDIS can work as a classical ID3. However, CONDIS, as well as the most widely known empirical inductive learning algorithms, including ID3 [30], AQ [25], and CART [5], is only able to learn "flat" concepts, uni-dimensional concept descriptions, or "labels": 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 simultaneously the problems of discriminating collisions from

obstructions and normal situations and discriminating between different types of collisions.

C. Learning of Taxonomic Knowledge

One extension could be learning multidimensional concept descriptions, but, having as motivation the automatic construction of the models required for the Assembly Supervisor, the idea of generating a concept hierarchy became more attractive. A new algorithm, SKIL (structured knowledge generated by inductive learning), was developed to perform that task [37]. SKIL requires the following specification of the application domain:

The concepts in the hierarchy are characterized by a set of classification attributes: $A = \{A_i{:}\, i = 1 \cdots a\}$. Each attribute can take one of a set of discrete values: Values$(A_i) = \{A_{ij}{:}\, j = 1 \cdots v_{Ai}\}$. The structure of the most abstract concepts in the hierarchy will be given by attributes selected from a set of top-level (start) attributes (TLA $\subseteq A$), which must be provided by the user.

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 be considered 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:

$$(\texttt{failure\_type}, \texttt{collision}, \{\texttt{obj\_size}, \texttt{obj\_hardness}, \texttt{obj\_weight}\}).$$

The values of the attributes of the concepts in the hierarchy are determined inductively based on training data specified in terms of a set of discrimination attributes or features: $F = \{F_i{:}\, i = 1 \cdots f\}$. When features are continuous, qualitative values are obtained in each step of the induction using the method described in Section IV-A-1. For discrete features, the set of values is provided by the user: Values$(F_i) = \{F_{ij}{:}\, j = 1 \cdots v_{Fi}\}$.

Each example in the training set is composed of a list of attribute-value pairs followed by a vector of feature values. For instance:

```
behavior = failure
failure_type = collision
collision_type = front
part_status = moved
-1  4  20  1  4  1  13  -1  0  1  0  2  -1
   13  1  -24  -71  -3  -15  -3  0.
```

In this case, the presented features were extracted from a trace of forces and torques.

The algorithm (see Fig. 14) 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. The first step is to verify which attributes can be closed, i.e., which attributes have the same value in all provided examples. In traditional inductive algorithms, this step corresponds to determining a class and creating a leaf node. In SKIL, determining the value of one or more classification attributes implies, by definition, the creation of a taxonomy node. If there are attributes whose values cannot be determined at the current stage, referred to as open attributes, induction continues. To be noted is the fact that, starting in the list of closed attributes and using the enabling triples, new open and closed attributes will be found recursively.

```
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) = SelectTestFeature(LEx,NewLAt,LFt);
if (FeatureIrrelevance(LEx,At,TF) > MAX_IRREL) {
    //MAX_IRREL: Max. feature irrelevance, e.g. 97.5%
    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. 14. The SKIL Algorithm.

In each stage of the induction, the main goal is to close classification attributes. For each attribute, the discrimination power of features is evaluated, in terms of an entropy measure, as in ID3 [30]. The feature that, for some attribute, gives the lowest entropy is selected to be test feature. If the chi-square test for stochastic independence [30] returns a confidence factor on the irrelevance of the test feature (concerning attribute value discrimination) greater than some threshold (e.g., 97.5%), expansion is stopped and a leaf node is created.

The basic knowledge transmutation used by SKIL is, therefore, empirical inductive generalization, only that at multiple levels of abstraction (see the inferential theory of learning [26]). 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. It is equivalent to a set of rules of the form

$$\forall x: \bigwedge_i F_i(x, fv_i) \Rightarrow \bigwedge_j A_j(x, av_j).$$

The left hand side of the implication is a conjunction (indicated by the $\wedge$ sign) of conditions, on the values of several discrimination features, sufficient to recognize the concept specified, on the right hand side, by a conjunction of attribute values.

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 [27]. This means that, for instance, the structure of the taxonomy in Fig. 13 would have to be user-defined. However, this is not flexible enough. Fixed decompositions have also been used for feature values [27], [33]. In the case of numeric features, since SKIL performs the clustering of numeric values in every decision node, the resulting decomposition tends to be the most adequate. In what concerns symbolic features, a decomposition of values could help. However, this was not implemented, since most of the available sensor data in the application that motivated the research is numeric. Developed with a particular problem in mind, SKIL is a contribution to the research in multilevel learning.

D. Training or Tutoring Methodology

According to the paradigm of programming by human demonstration, 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 corresponding monitoring knowledge. In the existing implementation, for each action and each continuous feature, the typical behavior of the attribute 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. 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, currently using SKIL.

When a new failure is detected during real execution of the assembly system, the human operator is called to classify and explain the 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 performance of the system.

V. Experimental Developments

A. Cell Integration

Experimental evaluation of the methodologies described above has been taking place in NovaFlex, a FMS/FAS pilot unit installed at the UNINOVA institute, and in the B-LEARN assembly cell, installed at Universidade Nova de Lisboa (UNL).

NovaFlex was conceived as a demonstration unit able to handle a set of typical activities of a computer integrated manufacturing (CIM) system [2]. Besides the machining and the assembly subsystems, the Pilot Unit includes a storage component, an input section for raw materials, a delivery section for finished products and a transportation subsystem that links all the other components. The transportation medium is a pallet-based conveyor belt. Each pallet can be adapted to transport different kinds of parts and products.

The system was required to comply with a variety of products (a basic design goal). The objective was to build a relatively generic infrastructure, that could adapt to a range of products with minimal setup effort. Another very important aspect is the possibility of different groups of users being simultaneously using different subsystems of NovaFlex for separate experiments. As a matter of fact, this situation is expected to be the most common in practice. This requirement led to an architecture in which NovaFlex can be operated either as an integrated FMS/FAS system or as a set of isolated subsystems (machining, assembly, transportation, storage, etc.). This has particular consequences on the design of the control architecture.

Therefore, the need to support these different research areas implied the design of a flexible architecture, from the topology to the control points of view. An easy reconfiguration of its operating mode is an important requirement to support concurrent research activities.

Fig. 15 illustrates the approach that was followed in NovaFlex. As mentioned in Section III, the set of methods of the controller model implement the actions that are needed to send the right commands to the real controller. The real controller image is developed using a client-server approach. In this way, implementation methods can ask this server to perform the required actions. These methods hide the underlying hardware structure from the application, i.e., an application using a robot component does not need to know much about the real controller and its image or server. The applications only know which functionalities are provided by the robot component model. This approach appears to be suitable to integrate existing controllers, making the integration of legacy systems an easier task.

In the B-LEARN assembly cell, experiments concerning the application of machine learning techniques in assembly supervision have been performed, in the framework of the European ESPRIT project B-LEARN. This cell is composed of one SCARA robot (needless to say, in all our robots, the control languages are not suited to write intelligent control software), three robot grippers, tools magazine and corresponding tool exchange mechanism, two special purpose feeders, one fixture

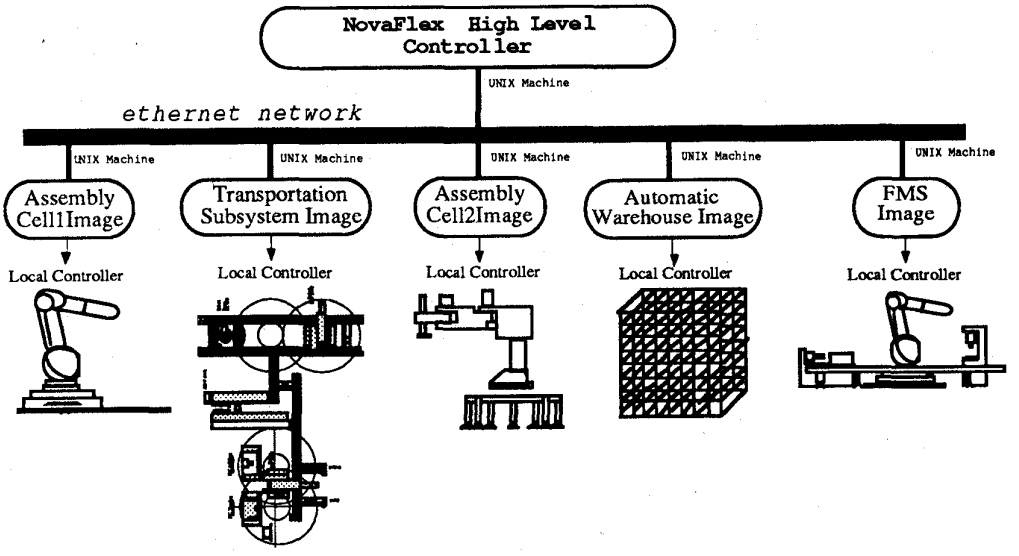


Fig. 15. NovaFlex physical infrastructure.

and sensing devices. As feedback information sources for the supervision system, the following discrete information sensors were integrated in the cell: a) in each gripper, to detect if it is open, closed or clamping; b) in feeders, to detect part presence, part stock existence and feeder problems; c) in the wrist of the robot, to find out which tool is attached, if any; d) in each tool place to detect tool presence; and e) in the fixture, to detect if the jig which will hold the assembly is present. The most frequent execution failures are expected to be those in which the robot arm is involved, including collisions, obstructions, and handling failures. Therefore, a force and torque sensor, which seems a good candidate to give information about those failures, was also included.

From the supervision point of view, the main limitation of the robot control language is that it does not provide guarded movements. Since communication via teach pendant (TP) is very fast, the solution was to decompose each motion command into a series of increments, executed sequentially via TP, until some condition is verified [40]. A server process emulating the teach pendant (TP emulator) and running on a dedicated PC, due to the tight communication cycle, was first developed. A layer added to the TP emulator provides guarded movements, other robot commands normally available via TP, and commands of other cell resources, that are actuated via the robot controller outputs. The TP emulator also provides information about robot errors. To the TP emulator plus the adaptation layer built on top of it, we call it an operational server.

As it is not easy to make acquisition of large quantities of sensorial data in UNIX workstations, and, on the other hand, concurrency in UNIX affects the sensor sampling rate, a program called low-level monitor is run in another dedicated PC, where it is quite simple and cheap to implant a data I/O board. The low-level monitor (LLM) checks conditions during the execution of actions, as specified by its client (the intelligent supervisor), and is able to answer questions about the state of the system during the diagnosis phase. The main services provided by the LLM are as follows.

- DEFINE_CONDITION—Define a sensory condition to be evaluated on demand.
- DEFINE_PROFILE—Define a sensory profile, i.e., a specification of allowable conditions for a set of sensor variables along a given time interval.
- START_EVALUATION—Start evaluating a previously specified condition. If a deviation is detected, the current action must be interrupted and the diagnosis function is called.
- STOP_EVALUATION—Stop evaluating a condition.
- MONITOR_PROFILE—Monitor a previously specified profile. In case of a deviation, execution is stopped and the diagnosis function is called.CONSULT_SENSOR—Read the value of a sensor.
- GET_BEHAVIOR—Return the behavior of all testable sensors during the execution of a terminated action.

Communication between the LLM and the Intelligent supervisor is accomplished via an RS232C line. Communication between the operational server (OS) and the intelligent supervisor is accomplished via RPC's (remote procedure calls). This infrastructure (Fig. 16) was developed using a client-server approach. The "recovered" controller includes the legacy controllers (robot, F&T sensor, etc.) and the LLM and OS server processes, runing on two dedicated PC's [40].

B. Learning of Diagnosis Knowledge

1) Training Situation and Training Data: The experimental work that will be presented concerns the identification of the execution failure, i.e., the classification part of the diagnosis process. Some failures can easily be identified by simple discrete sensors. For instance, if the wrong tool is attached to the robot, that situation can be detected by one sensor. If the part is missing in feeder, that may as well be detected with little effort. Such kind of knowledge can be easily coded by hand as rules.

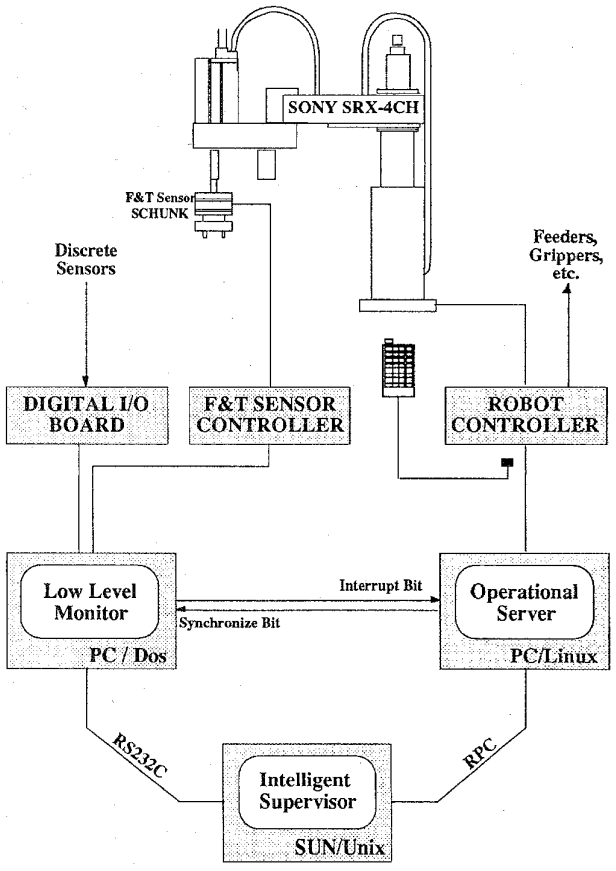


Fig. 16. Assembly cell integrating infrastructure (B-LEARN project).

However, a question remains: How to characterize the situation in which the force profile in the robot wrist is not normal? Different external exceptions can occur causing execution failures that manifest through abnormal force and torque profiles. These profiles, although sometimes recognizable by the human, are difficult to model analytically. Therefore, what would be desirable is that the system learned to look at the force profiles in order to identify different situations.

The chosen case study is the macrooperation ≪Pick and Place≫ of a part, which can be hierarchically decomposed as shown in Fig. 17. For the experiments, we selected three of the primitives involved in the operation: a) approach to grasp position (`Approach-Grasp`); b) `Transfer` (of part); and c) approach to the final position (`Approach-Ungrasp`). During the training phase, each of the selected operations was executed many times and several external exceptions were provoked. In most cases an object was placed, either in motion or stopped, in the robot arm motion path. The forces and torques trace in an interval surrounding each failure was collected and the failure classification was associated to it. The length of the trace is of 15 samples.

In this way, for the operation `Approach-Ungrasp`, 117 classified examples were collected. The following failure classes were considered (see force profiles in Fig. 19): 0: normal behavior; 1: collision in part and part moved; 2: collision in part and part lost; 3: collision in tool; 4: front collision and part moved; 5: front collision and part lost; and 6: obstruction. For the operations `Approach-Grasp` and `Transfer`, less examples were collected (88 and 47, respectively), but more information about the failure situation was recorded. This information is organized in terms of the following attributes.

- *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,* etc. (see Fig. 18).
- *object size*—size of object causing failure: *small, large.*
- *object hardness*—can be *soft* or *hard.*
- *object weight*—can be *low* or *high.*

2) Experimental Results: To run learning algorithms, some preprocessing of the raw sensor data is needed. In fact, if all numerical values in a force or torque trace, in total 15 values, are given to the learning algorithm, it will probably run less efficiently and the knowledge produced will be less readable and less efficient to use. On the other hand, when humans look at force profiles, they can easily recognize trends and high level features that the learning algorithm will ignore if the training set is not given to it in terms of the "good" features. For these experiments, using measures such as average, slope, and monotonicity, higher level features were extracted from raw sensor data (same method as in [11]). In this way, for each force or torque profile, we reduced the total number of features from 15 to 7, being 4 of them (slopes and monotonicity) clearly of a higher level of abstraction.

One of the goals of the performed experiments was to evaluate the impact of the method for signals to symbols conversion, presented in Section IV-A-1, comparing it with "blind" discretization. For this problem, the CONDIS learning kit was used on the data collected during the execution of the `Approach-Ungrasp` primitive. The following two implementations of the rule *Transform Features* (R5) were provided to the system.

- R5.1: "Blind" discretization: the domain of values of continuous features is transformed into a set of intervals of equal length that are used as discrete values. The number of intervals is given by the Number of Struges: $1 + \log 2N$, where N is the number of examples.
- R5.2: The domain of values of each continuous attribute is transformed into a set of qualitative values following the method described in Section IV-A-1.

For feature selection, an entropy based rule, like in ID3, was used. Partitioning examples in a node for further expansion is also done as in ID3: create a branch for each value of the test feature of the current node. Two different implementations of the rule *Test Stop* (R1: for deciding when to create a leaf node) were also provided.

- R1.1: Create a leaf node when all examples belong to the same class, or when the current list of attributes is empty.

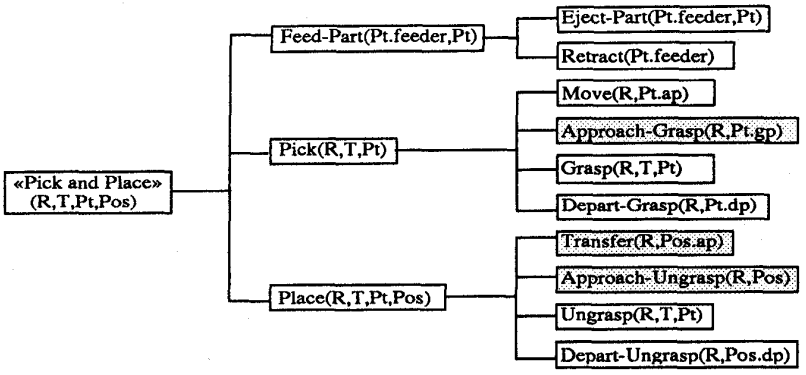


Fig. 17. Hierarchical decomposition of a ≪Pick and Place≫ macrooperation.

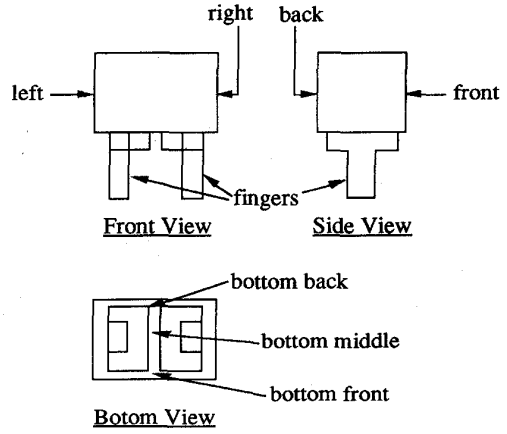


Fig. 18. Regions of a gripper that may be involved in a failure.

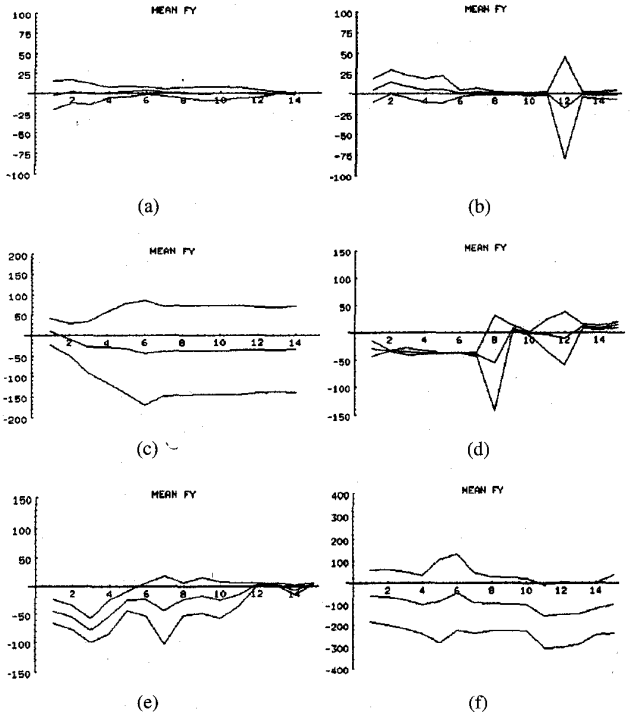


Fig. 19. Typical behavior of force Fy during different types of failures in the Approach with part operation. (a) Collision in part and part moved. (b) Collision in part and part lost. (c) Collision in tool. (d) Front collision and part lost. (e) Front collision and part moved. (f) Obstruction.

- R1.2: Create a leaf node in the conditions of R1.1 or if the chi-square test for stochastic independence [30] returns a confidence on the irrelevance of the best feature higher than 97.5%.

As can be seen from the Table I, the largest tree, corresponding to the simultaneous use of rules R1.1 and R5.1, has 108 nodes: 63 leaf nodes that represent the learned rules, and 45 interior nodes that represent the points of decision. An example of a learned rule is

```
if (μFz2 in [−361.1,−268.4[) and
   (μFx1 in [−64.6,−63.4[)
   then behavior = "front collision with
   part lost";
```

As it was expected, rule R1.2, which stops branching when the irrelevance of the test feature is too high, reduces the number of decisions to 29. The number of rules is preserved and therefore the global number of nodes is reduced from 108 to 91. On the other hand, rule R5.2 reduces the number of decisions from 45 to 35 and the number of rules from 63 to 56. The global number of nodes is reduced from 108 to 92.

The smallest tree is produced when rules R1.2 and R5.2 are used simultaneously. In that case, the number of decisions is reduced to 16 and the global number of nodes is reduced to 72.

TABLE I
RESULTS OF APPLYING CONDIS ON THE DATA OF Approach-Ungrasp

R1: Test Stop	R1.1	R1.1	R1.2	R1.2
R5: Transform Feature	R5.1	R5.2	R5.1	R5.2
Number of Tree Nodes	108	91	92	72
Number of Decisions	45	35	29	16
Number of Rules	63	56	63	56
Decisions per Rule	0.71	0.63	0.46	0.29
Error Rate	40%	46%	40%	46%

The rule R5.2 produces good results since the tree becomes more concise. When using R1.1, the reduction in the number of nodes produced by R5.2 is of 16%. When using R1.2, the

TABLE II
APPROACH-UNGRASP PROBLEM SPECIFICATION

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

(a)

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

(b)

reduction in the number of nodes produced by R5.2 is of 22%. However, rule R5.2 seems to lead to slightly higher error rates. This is a problem that must be better investigated. In any case, the gain in simplicity of the generated tree seems to be greater than the loss of accuracy. In the four experiments, the error rates (leave-one-out test) were very high, around 40% to 46%. This is due, mainly, to not having enough training examples.

In the `Approach-Ungrasp` problem, the failure classifications have embedded some sort of hierarchy. For instance, there are three major types of collisions: collision in part, collision in tool and front collision. Some of these still have more refined descriptions. This implicit hierarchy could be used to guide the induction process and possibly reduce the error rates. That is what will be attempted next, using SKIL.

The concept hierarchy that this algorithm learned characterizes the execution situation at different levels of detail. The most detailed descriptions will correspond to the seven failure classifications considered above. The set of classification attributes (in the SKIL sense) shown in Table II(a) seems to be enough to obtain the most detailed descriptions. The attribute enabling statements are shown in Table II(b). The top-level (start) attributes are *behavior* and *part_status*. The discrimination attributes or features are the same as before, and the same preprocessing was applied. The classifications in the table of examples were decomposed according to the classification attributes.

After running SKIL on the new domain specification and new table of examples, a decision tree was obtained having 71 nodes. The concept hierarchy contained in the tree has 59 nodes, being 10 of them internal nodes and 49 terminal nodes (see Fig. 20). Examples of the corresponding rules are as follows:

```
∀ x: (Fz1(x, [7,21[) & Fx1(x, [-4.5,1[)
          & Dx1(x, [-0.5,0.5[))
     ⇒ ( behavior(x,normal)
          & part_status(x,ok))
∀ x: (Fz1(x, [-995,7[)
          & Dz2(x, [-542,-51[)
          & Fx3(x, [-464,-13[))
     ⇒ ( behavior(x,failure)
          & part_status(x,moved)
          & failure_type(x,obstruction)).
```

The size of the tree is very similar to the size of the tree generated by CONDIS using rules R1.2 and R5.2 (72 nodes). This was expected since, in SKIL, numerical to symbolic conversion is done by the same method as in R5.2. SKIL also uses the chi-square test for stochastic independence as in R1.2.

Performing the leave-one-out test with the same data and algorithm, the resulting average error rate is 15%, much lower than in the "flat" classification obtained in any of the four experiments with CONDIS (see Table III).

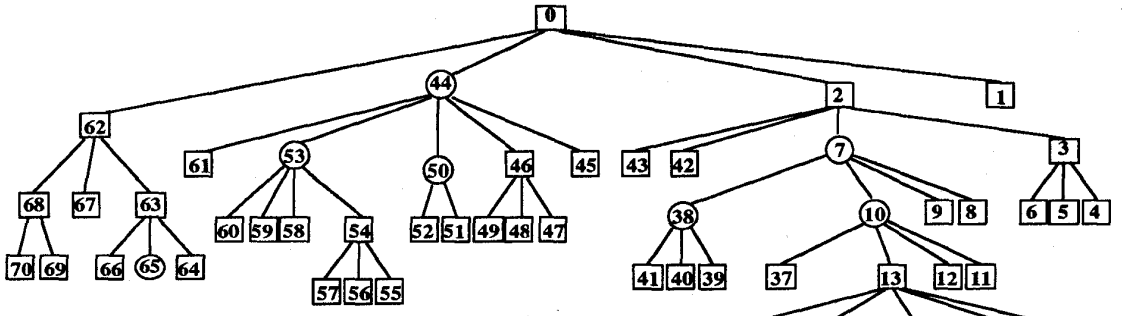
From this comparison we see that SKIL may be used to generate more accurate knowledge. However, its great advantage is that it is able to generate conceptual hierarchies. The problem of generating failure classification knowledge for the `Approach-Ungrasp` primitive was initially formulated in terms of seven classes of failures. Then, the problem was reformulated for SKIL in terms of four classification attributes. The total number of complete failure descriptions that can be built using the attributes and their values is 15. Of course, some of them never occur (e.g., $\{\text{behavior} = \text{normal}, \text{part\_status} = \text{lost}\}$), and others were not present in the training set.

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 failures of `Approach-Grasp`, which included 10 attributes, 28 values and 8 enabling triples (see domain specification on Table IV).

The characteristics of the decision tree and concept hierarchy generated by SKIL starting with top-level attribute *behavior,* are shown in Table V. The global number of nodes is 93. The error rate (30%) is much higher than in the previous problem when SKIL was also applied (15%). This is understandable since the target concept is much more complex and a smaller training set was provided (only 88 examples). For the `Transfer` problem, for which only 47 examples were collected, a taxonomy was also generated by SKIL, and the error rate was 34%. We see, as a general trend, that as the number of occurrences of each attribute value in the training set increases, the corresponding error rate decreases (Fig. 21).

The general approach is, therefore, to collect examples of normal and abnormal behavior of each operation or operation-type/operator and generate a behavior model (Fig. 20) that the diagnosis function (Figs. 3 and 4) will use to verify the existence of failures, to classify and explain them and to update the world model. The developed methodologies and the performed experiments are a contribution to the failure classification part of the diagnosis task. Failure explanation is a topic for further research.

The CONDIS learning kit and the algorithm SKIL are tools that can be easily integrated in a performer and used to generate knowledge from examples. With CONDIS we empirically demonstrated the viability of our approach concerning the signals to symbols conversion. The results obtained with SKIL seem rather promising since it produces structured (taxonomic) knowledge with a higher degree of accuracy.



□ **Taxonomy Node**
○ **Decision Tree Node**

Learned descriptions of the target concept at different levels of detail:

Node 0 : { } ;
Node 2 : { behavior = failure } ;
Node 3 : { behavior = failure, part_status = moved } ;
Node 4 : { behavior = failure, part_status = moved, failure_type = obstruction } ;
Node 13 : { behavior = failure, failure_type = collision } ;
Node 24 : { behavior = failure, failure_type = collision, collision_type = front, part_status = moved} ;
Node 51 : { behavior = normal, part_status = ok } ;
....

Fig. 20. Behavior taxonomy generated by SKIL for the `Approach-Ungrasp` primitive.

TABLE III
SKIL VERSUS CONDIS ON THE DATA OF `Approach-Ungrasp`

Algorithm	CONDIS	SKIL
Number of Tree Nodes	72	71
Number of Taxonomy Nodes	—	59
Number of Taxonomy Leafs	—	49
Number of Taxonomy Interior Nodes	—	10
Number of Decisions	16	21
Number of Rules	56	50
Number of Decisions per Rule	0.29	0.42
Error Rate (Leave-one-out)	46%	15%

TABLE IV
ATTRIBUTES, VALUES, AND ENABLING TRIPLES

Attribute	Attribute Values
behavior	{ normal, failure }
phase	{ initial, middle, terminal }
failure_type	{ collision, fr_collision, obstruction }
affected_body	{ tool, tool_tubes, fingers }
tool_region	{ front, right, left, back, bottom }
fingers_region	{ left, right, both }
bottom_subregion	{ front, middle, back }
obj_size	{ small, large }
obj_hardness	{ soft, hard }
obj_weight	{ low, hight }

(a)

Attribute	Value	Enabled Attributes
behavior	failure	{ failure_type, affected_body }
behavior	normal	{ phase }
failure_type	collision	{ obj_size, obj_hardn, obj_weight }
failure_type	fr_collision	{ obj_size, obj_hardn, obj_weight }
failure_type	obstruction	{ obj_size, obj_hardn, obj_weight }
affected_body	tool	{ tool_region }
affected_body	fingers	{ fingers_region }
tool_region	bottom	{ bottom_subregion }

(b)

Accuracy, however, is a problem requiring further investigation. Work, in the context of the European ESPRIT project B-LEARN, in cooperation with the University of Turin, in which the learning tool Smart+ [4] was applied to the same data, could not solve the accuracy problem either. If the lack of examples, which are expensive to acquire, was one of the main causes for the less satisfactory results concerning accuracy, this implies that more research effort must be put in the design of the training methodology. The continuation of the research focused on efficient ways of collecting examples (including example interpolation), in feature construction and selection and in long-term learning, and some interesting results have been reported [38], [39].

VI. CONCLUSION AND FUTURE WORK

Planning and executing flexible assembly tasks on a real industrial environment is a highly complex problem that must take into account (benefit from) the multiple inter-relationships with other activities involved in the manufacturing process. In particular, the design of an intelligent execution supervisor needs a clear understanding, not only of the execution planning phase but also of other phases like process planning, product design, scheduling, production planning, etc.

An evolutive execution supervision architecture was presented in this framework. Flexibility implies increasing the on-line decision making capabilities, for which dispatching, monitoring, diagnosis and error recovery functionalities have been devised. The lack of comprehensive monitoring and diagnosis knowledge in the assembly domain points out to

TABLE V
EVALUATION OF KNOWLEDGE GENERATED BY SKIL ON THE DATA OF Approach-Grasp

Number of Tree Nodes	93
Number of Taxonomy Nodes	86
Number of Taxonomy Leafs	62
Number of Taxonomy Interior Nodes	24
Number of Decisions	31
Number of Rules	62
Number of Decisions per Rule	0.50
Error Rate (Leave-one-out)	30%

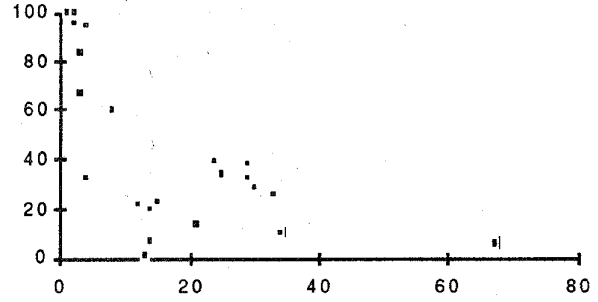


Fig. 21. Error rates per attribute value according to the number of examples provided (Approach-Grasp problem).

the use of machine learning techniques, leading to an evolutive architecture. Preliminary experiments in this direction demonstrated the feasibility of the approach, and allowed the identification of the main difficulties and following steps.

Finally, another important aspect in any integrated approach to the assembly supervision problem is the integration of legacy systems. A methodology developed in this work proved successful as a method to integrate existing device controllers into an open infrastructure.

ACKNOWLEDGMENT

The authors would like to thank J. C. Silva for his contribution to the experimental setup.

REFERENCES

[1] A. P. Ambler, S. A. Cameron, and D. F. Corner, "Augmenting the RAPT robot language," in *Proc. NATO Advanced Research Workshop on Languages for Sensor-based Control in Robotics,* NATO ASI series, 1987, no. 29, pp. 305–316.

[2] J. Barata and L. M. Camarinha-Matos, "Development of a FMS/FAS system–The CRI's pilot unit," *Studies Informatics Contr.,* vol. 3, no. 2–3, pp. 231–239, 1994.

[3] J. Barata, L. M. Camarinha-Matos, and J. F. Rojas Chavarria, "Modeling, dynamic persistence and active images for manufacturing processes," *Studies Informatics Contr.,* vol. 3, no. 2–3, pp. 173–183, Sept. 1994.

[4] M. Botta and A. Giordana, "SMART+: A multi-strategy learning tool," in *Proc. Int. Joint Conf. Artificial Intelligence (IJCAI-93),* 1993, pp. 937–943.

[5] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees.* Belmont, CA, Wadsworth Int. Group: 1984.

[6] L. M. Camarinha-Matos, "Sistema de programação e controle de estações robóticas. Uma arquitectura baseada em conhecimento," Ph.D. dissertation, Universidade Nova de Lisboa, June 12, 1989.

[7] L. M. Camarinha-Matos and H. Afsarmanesh, "Federated information systems in manufacturing," in *Proc. European Robotics and Intelligent Systems Conf. (EURISCON'94),* Malaga, Spain, 1994, pp. 1598–1613.

[8] L. M. Camarinha-Matos, U. Negreto, G. R. Meijer, J. Moura-Pires, and R. Rabelo, "Information integration for assembly cell programming and monitoring in CIM," presented at the 21st ISATA, Wiesbaden, Germany, 1989.

[9] L. M. Camarinha-Matos and A. L. Osório, "Monitoring and error recovery in assembly tasks," presented at ISATA, Vienna, Austria, 1990.

[10] L. M. Camarinha-Matos and H. J. Pinheiro-Pita, "Interactive planning of motion and assembly operations," in *Proc. IEEE Int. Workshop Intelligent Motion Control,* Istambul, Turkey, 1990, pp. 587–592.

[11] L. M. Camarinha-Matos, L. Seabra Lopes, and J. Barata, "Execution monitoring in assembly with learning capabilities," *Proc. 1994 IEEE Int. Conf. Robotics and Automation,* San Diego, CA, pp. 272–279.

[12] S. Chakrabarty and J. Wolter, "A hierarchical approach to assembly planning," in *Proc. 1994 IEEE Int. Conf. Robotics and Automation,* San Diego, CA, 1994, pp. 258–263.

[13] J. Cheng, U. M. Fayad, K. B. Irani, and Z. Qian, "Improved decision trees: A generalized version of ID3," in *5th Int. Conf. Machine Learning,* 1988, pp. 100–106.

[14] J. de Kleer, A. K. Mackworth, and R. Reiter, "Characterizing diagnoses," in *8th National Conf. Artificial Intelligence (AAAI-90),* Boston, MA, 1990, pp. 324–330.

[15] K. Forbus, "Qualitative physics: Past, present and future," *Exploring Artificial Intelligence,* H. Shrobe, Ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 239–296.

[16] B. Furth, *Automated Process Planning,* NATO Advanced Study Institute on CIM: Current Status and Challanges. Berlin-Heidelberg: Springer-Verlag, NATO ASI Series, 1988.

[17] Gespac International SA, *FIELDBUS Hardware and Firmware Specification Manual,* 1992.

[18] M. Gini, "Symbolic and qualitative reasoning for error recovery in robot programs," *NATO ASI Ser., Lang. Sensor-Based Contr.,* vol. F29, pp. 147–167, 1987.

[19] K. Hirota, Y. Arai, and S. Hachisu, "Moving mark recognition and moving object manipulation in fuzzy controlled robot," *Contr.-Theory Adv. Technol.,* vol. 2, no. 3, pp. 399–418, 1986.

[20] L. S. Homem de Mello and S. Lee, Eds., *Computer-Aided Mechanical Assembly Planning.* Boston/Dordrecht/London: Kluwer, 1991.

[21] ISO/IEC 9506, manufacturing message specification, (1: service definition; 2: protocol definition), 1990.

[22] E. López-Mellado and R. Alami, "A failure recovery scheme for assembly workcells," *1990 IEEE Int. Conf. Robotics and Automation,* Cincinnati, OH, 1990, pp. 702–707.

[23] MAP/TOP Users Group, *Manufacturing Automation Protocol Specification, V3.0,* 1991.

[24] G. R. Meijer, "Autonomous shopfloor systems—A study into exception handling for robot control," Ph.D. dissertation, Universiteit van Amsterdam, Amsterdam, The Netherlands, 1991.

[25] R. S. Michalsky, "On the quasi-minimal solution of the general covering problem," in *Proc. 5th Int. Symp. Information Processing,* Bled, Yugoslavia, 1969, pp. 125–128.

[26] ——, "Inferential theory of learning: Developing foundations for multistrategy learning," *Machine Learning. A Multistrategy Approach,* vol. IV, R. S. Michalsky and G. Tecuci, Eds. San Mateo, CA: Morgan Kaufmann, 1994.

[27] I. Mozetic, "The role of abstractions in learning qualitative models," in *Proc. 4th Int. Workshop Machine Learning,* Irvine, CA, 1987, pp. 242–255.

[28] ——, "Hierarchical model-based diagnosis," *Int. J. Man-Machine Studies,* vol. 35, pp. 329–362, 1991.

[29] A. L. Osório and L. M. Camarinha-Matos, "Support for concurrent engineering in CIM-FACE," *Balanced Automation Systems. Architectures and Design Methods,* L. M. Camarinha-Matos and H. Afsarmanesh, Eds. London: Chapman & Hall, 1995, pp. 275–286.

[30] J. R. Quinlan, "Induction of decision trees," *Machine Learning,* vol. 1, pp. 81–106, 1986.

[31] R. Rabelo and L. M. Camarinha-Matos, "Control and dynamic scheduling in virtual organization of production resources," *IFIP Transactions on Production Management Methods.* Amsterdam: North-Holland, 1994, pp. 359–368.

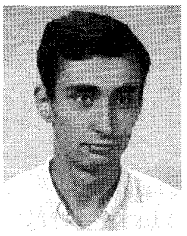
[32] C. Ramos, "Planeamento e Execução Inteligent de Tarefas de Montagem e de Manipulação (in Portuguese)," Ph.D. dissertation, Universidade do Porto, Portugal, 1993.

[33] Y. Reich, "Macro and micro perspectives of multistrategy learning," *Machine Learning. A Multistrategy Approach,* vol. IV, R. S. Michalsky and G. Tecuci, Eds. San Mateo, CA: Morgan Kaufmann, 1994, pp. 379–401.

[34] L. Seabra Lopes, "Golog 2.0 Um Gestor de Objectos em Prolog," Dep. Elect. Eng., Universidade Nova de Lisboa, Lisbon, Portugal, Tech. Rep. UNL 12-94, 1994.
[35] L. Seabra Lopes and L. M. Camarinha-Matos, "Learning in assembly task execution," in *Proc. II European Workshop Learning Robots,* Turin, Italy, 1993, pp. 129–142.
[36] ——, "Learning to diagnose failures of assembly tasks," in *Proc. Int. Conf. Artificial Intelligence in Real-Time Control,* Valencia, Spain, Oct. 1994, pp. 125–131.
[37] ——, "Inductive generation of diagnostics knowledge for autonomous assembly," in *Proc. 1995 IEEE Int. Conf. Robotics and Automation,* Nagoya, Japan, 1995, pp. 2545–2552.
[38] ——, "Planning, training and learning in supervision of flexible assembly systems," *Balanced Automation Systems. Architectures and Design Methods,* L. M. Camarinha-Matos and H. Afsarmanesh, Eds. London: Chapman & Hall, 1995, pp. 63–74.
[39] ——, "Example generation and processing for inductive learning in the assembly domain," in *Proc. 4th European Workshop on Learning Robots,* Karlsruhe, Germany, 1995, pp. 1–9.
[40] L. Seabra Lopes, J. C. Silva, and L. M. Camarinha-Matos, "Supervisão, Aprendizagem e Integração de Serviços em Robótica de Montagem," in *Proc. 5as J. Projecto, Planeamento e Produção Assistidos por Computador,* Guimarães, Portugal, 1995, pp. 397–404.
[41] G. Spur *et al.*, "Operational control for robot system integration into CIM," IPK, Berlin, Esprit 623, 5th Interim Rep., 1987.
[42] S. Srinivas, "Error recovery in robot systems," Ph.D. dissertation, Calif. Inst. Technol., Pasadena, 1977.
[43] K. Tierney, R. Browden, and J. Browne, "ESPFAS—A prototype expert system for technological planning in robot based flexible assembly systems," *Ann. OR,* vol. 15, pp. 111–130, 1986.

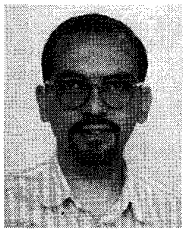
Luis M. Camarinha-Matos (M'92) received the Ph.D. degree in computer engineering from the Universidade Nova de Lisboa (UNL), Portugal, in 1989.

He is a Co-Founder of the Electrical Engineering Department of UNL where he is currently Auxiliary Professor (equivalent to Associate Professor) and Coordinator of the Robotics and CIM research unit. He has participated in many international and national projects, both as researcher and as coordinator. He has been involved in the organizing and program committees of various international conferences He has edited various issues of journals and books and has more than 90 publications in journals and conference proceedings.



Luis Seabra Lopes (S'92) graduated from the Computer Science Department of the Universidade Nova de Lisboa (UNL), Portugal, in 1990.

He is currently a doctoral candidate in the Electrical Engineering Department of UNL. He became a member of the Intelligent Robotics Group of the same university after receiving his degree in 1990. He is currently participating in the european ESPRIT project B-LEARN, one of the first projects to investigate applications of machine learning to robotics. At the moment, his main interests are robotics, machine learning, and intelligent supervision.



José Barata (M'87) received the computer science degree from the Universidade Nova de Lisboa (UNL), Portugal, in 1990, and the M.Sc. degree from the same University in 1995.

Currently, he is an Assistant at the Electrical Engineering Department of UNL and belongs to the Robotics and CIM research group. His mains interests are CIM systems integration and intelligent manufacturing systems.