Presented at: AAAI-94 Spring Symposium on Detecting and resolving Errrors in Manufacturing Systems, Stanford, Ca., USA.

Manufacturing Control System Principles supporting Error Recovery

Peter Loborg and **Anders Törne**
Dept. of Computer and Information Science,
Linköping University, S-58183 Linköping, SWEDEN
E-mail: petlo@ida.liu.se, andto@ida.liu.se

February 20, 1995

Abstract

This paper argues that manufacturing control systems should be structured to retain information used while specifying the system so it may be reused during error recovery situations. An example of such a system (the Aramis system) and its proposed error recovery capabilities is presented.

Introduction

Most Manufacturing Control Systems (MCS) commercially available lack support for error recovery. Even normal constructs for encoding exception handling, a natural part of many computer languages, are absent. Although several proposals have been suggested, there are still shortcomings:

- although many proposals contain support for error recovery or fault tolerant operation, most of the support only concerns the assembly plan/activities of the manufacturing process, such as what to do when a grasp operation fails (e.g. [Cao and Sanderson, 1992; Delchambre and Coupez, 1988; Gaspart *et. al.*, 1989]). However, after a structural failure such as a broken air pressure hose, and subsequent repair, parts of the machinery may be in an abnormal state (i.e. not passed during normal execution), or in a normal state but not corresponding to the 'state' of the program execution - a situation seldom addressed.
- they are often tedious to instruct, not to mention the effort needed to modify the system. One problem is that the knowledge used when instructing a MCS is to a large extent lost at coding/compile time - i.e. it is not explicitly needed/represented in the systems normal instruction formalism, and thus not available for further usage. As a result of this, many earlier attempts to provide support for error recovery often result in multiple representation of the knowledge (e.g. [Lee *et. al.*, 1983; Schmidt, 1992; Srinivas, 1977; Srinivas, 1978; Taylor and Taylor, 1988; Taylor *et. al.*, 1990), implying extra overhead when the system changes and the knowledge base is to be updated. A prime example is the usage of an external expert system responsible for detecting anomalies and for producing recovery actions, hooked on to an existing controller. A small change of the program in the controller results in major rewritings of the expert system. However, there are approaches where knowledge is reused (e.g. [Gini, 1983]) or integrated as a part of the instruction formalism ([Chen and Trivedi, 1991; Delchambre and Coupez, 1988; Gaspart *et. al.*, 1989; Meijer and Hertzberger, 1988; Meijer *et. al.*, 1991]).
- the competence of the user of these systems, i.e. the operator at the shop floor, is seldom reused. Most proposals presented are designed to solve the problem automatically, thus limiting their applicability.

The area of error recovery may be subdivided in detection, diagnose and recovery. We are primarily interested in providing support in the recovery phase, or more precisely, providing support to change the state of the machine to be a legal one, and to synchronize the state of the current execution with the state of the machine. The goal is to minimize the time and material loss during the recovery process.

The means are twofold: Firstly, the usage of an explicitly represented model of the underlying system, providing robust service for a task level instruction system as well as a semantic framework for any planning or plan repair activity. Secondly, a task level where the activities, their causal relation and what object or type of object they use are logged in order to extract information about the normal behavior of the system.

The next section presents an overview of Aramis (A Robot And Manufacturing Instruction System), which is a specification, programming and execution environment for manufacturing applications. The following section describes the recovery support (which will be[1]) provided by the system.

1. The programming environment exists as described herein, used in a robot cell equipped with vision, tactile sensors etc. [Loborg *et. al.*, 1994; Loborg *et. al.*, 1993; Loborg and Törne, 1991; Törne, 1990]. The error recovery support as presented herein is under development, and will be presented in a forthcoming licentiate thesis.

Aramis - An Overview

The Aramis system has been designed with a layered architecture, based on different levels of abstraction. It consists of three different levels; task programming level, the control level and the physical level (Fig.1). It uses a *world model* (WM) as interface between the task level and the control level.

At the task level the operator specifies what operations or actions should be performed upon the WM and under what conditions, using a graphical hybrid rule based language [Loborg and Törne, 1991]. The physical environment is modelled as a set of objects in the WM, where each object has a set of state variables. The task program executes by setting reference values for the objects (their state variables), thereby requesting the objects to change their state. The control level acts as a servomechanism and is responsible for keeping the real world in a state represented in the WM, as the WM reference values are changed by task program execution.

The programming at the control level is typically done by control engineers, and consists mainly of specifying how devices in the real world should be viewed in the WM, and what control algorithms is needed to implement the servo mechanism.

The physical level is the actual connection to the real world, where explicit I/O, requested by control level algorithms, is performed with sensors and actuators.

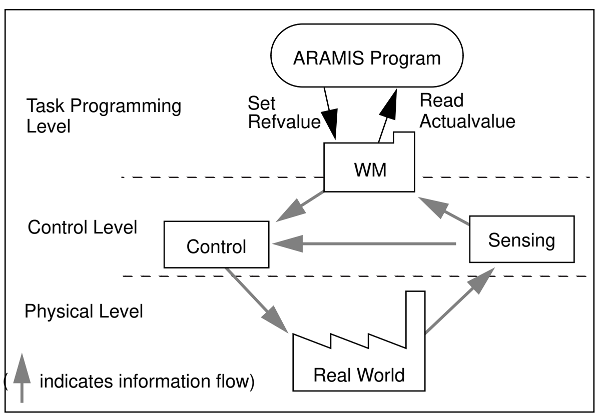

Figure 1 The world model is viewed as a set of reference values which is set by the Aramis program and used by a servo controller (sensing/control) to invoke correct control algorithms in the 'real world'. The WM also support the reading of actual values.

The remainder of this section is devoted to the world model (how objects are modelled, etc.), the control level and its connection to the real world, and finally there is a brief presentation of the task level language.

World model based programming

In Aramis, we adopt an object centered view of the world, where each object is responsible for its own internal state and its consistency. The task level program is responsible for coordinating the operation of all objects involved.

Each object contain an internal *state*[1] and information about its proper usage. This will prevent the task level programmer from abusing the object unnoticed - an exception will occur. This principle is best explained through the following example, where a two speed bidirectional electrical engine is modelled (fig 2). This device is represented in the WM by an object containing two state variables, named `speed` and `direction`. The domain of `speed` is a set of the three symbols `stop`, `low` and `high`. The domain of `direction` consists of `forward` and `backward`. This results in an object with six possible states. We use a set of *constraints* to group the state space into a set of *interesting* states (e.g. when the value of `speed` equals `stop`, the value of `direction` is uninteresting, and thus five states is adequate to describe the engine). A *finite state machine* (FSM) defined over the set of interesting states is used to describe how these variable values may be combined and changed in this object. To each transition defined in the

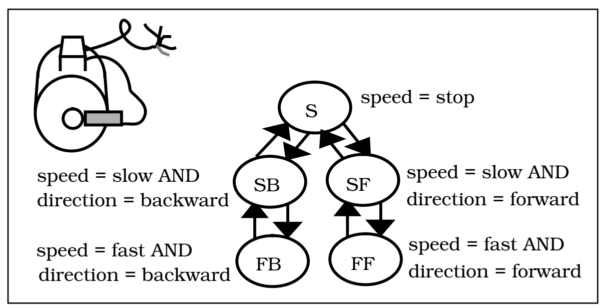

Figure 2 An electrical, bidirectional two speed engine is modeled by two variables, direction and speed. Its behaviour is modeled by a FSM with five interesting states, each representing a combination of variable values according to its constraint. The arcs between the states describe legal transitions, rouling out the posibility to reverse the engine without requesting it to stop in between.

FSM there might be an associated control algorithm responsible for achieving something in the real world which corresponds to the transition of states in the object. This functionality is part of the control level and will be detailed in next subsection.

The task level program controls the object by requesting changes of its variable values. The object will respond upon a request with one of the following:

1. We use the word *state* to denote a specific combination of values for a fixed set of variables. Each variable take its values from a finite (and thus discrete) domain.

- error - there is no direct path from the current state to any state representing the requested values, i.e. a state where the requested values will satisfy the constraint (task level programming error).
  Or: there are several direct paths to states corresponding to requested variable values (object modelling error - should never occur)
- failed - the object did not succeed in reaching the requested variable values. This may be due to an engine overload situation, some fault in the engine or the electrical hardware from the I/O board and onwards, or a programming error in the algorithm responsible for achieving the state. This reply also indicates that any built in recovery actions have failed.
- succeed - the new state is reached and the world has a state which corresponds to the requested state.

Each request from the task level program will be suspended until the object returns the result, in which case the execution of the requesting 'process' continues as normal or an exception/error is raised.

This model of an object is well suited for specifying aspects of real time behaviour, such as *transition completion time*[1] and *algorithm periodicity*. This is vital when designing controllers for any continuous physical process.

Control layer - implements objects

The control layer hardware consists of a set of computing devices and peripheral hardware organized in a topology of communication channels, which should provide real time computational power and I/O to the world model.

Although this has not been a part of the project, the modeling of this topology as well as the analysis of computational needs (based on the object models) and scheduling of the actual computation are topics that need to be addressed in order to build a complete system. Major contributions have already been made in the areas of real time communication systems and protocols (e.g. [Thomesse *et. al.*, 1991]), scheduling algorithms applicable both to communication and process scheduling (e.g. [Leinbaugh and Yamini, 1982; Rajkumar *et. al.*, 1988; Stancovic *et. al.*, 1990; Fohler, 1992; Sha and Sathaye, 1992]) as well as distribution of time with bounded accuracy (e.g. [Kopetz and Ochsenreiter, 1987; Kopetz, 1992]). The problem of allocating processor nodes for different sets of processes depending on both their computational and communicational need and the need for processor specific services (such as an I/O device) has also been investigated (e.g. [Ma *et. al.*, 1982; Verhoosel *et. al.*, 1991]).

Our concern has been to find a suitable specification for the computational needs that support both the scheduling algorithms and analytical methods used, and an abstraction/interface to a task level language.

The interface consists of a set of objects and their state variables, and a principle for how the task level interacts with objects. However, if the same abstraction of the physical world as presented by available sensors and actuators would be used at the task level, the resulting state space would be enormous as would the amount of communication needed to keep the world model updated in a distributed system.

Thus, each object type is described in two parts. Firstly an abstract model of the object is specified, as presented in previous subsection. Secondly, the implementation of it is specified. This includes the specification of what types of actuators and sensors are used, which also implies what data types are used at this level, code to extract information and control the device, etc.

These two parts are then connected by describing a mapping from selected variables or sensors of the object to the state variables describing the interface. Appropriate parts of the code defined in the object are associated with transitions and nodes of the FSM of the object - to be used to accomplish a transition or maintain a reached state.

Example: If the engine in Fig.2 is equipped with a speedometer producing values as integers in the range of 0..3000, the cardinality of the state space at the control level is 3001x 2. In the state space as viewed in the WM (presented in Fig.2) the variable speed is reduced to range over 3 values, and thus the state space is reduced to a cardinality of 6. This state space is then grouped into 5 *interesting* states, over which a behaviour is defined in terms of legal transitions, constituting a FSM. A change of value of the speedometer sensor at the control level is not propagated to the world model until it implies a change in the corresponding WM-variable of the object.

Formally, the state space as viewed by the sensors and actuators available (S_{RT}) is reduced or mapped onto an *exported* state space (S_E - the one present in or exported to the WM) by a surjective function $f: S_{RT} \rightarrow S_E$. Therefore all low level states have a mapping. The same mapping also denotes an inverse, injective function $f^{-1}: S_E \rightarrow S_{RT}$ implying that state changing *requests* issued at the task level have a unique counterpart at the control level. Whether the control algorithms responsible for achieving a state change in S_{RT} such that $f^{1}(s_{E\ request})$ equals $s_{RTresult}$, the resulting state, actually will do that or not is implementation dependant. What is required is that $f(s_{RTresult})$ equals $s_{E\ request}$. The constraints used to specify the mapping from the exported state space to the set of interesting states, also called the *abstract* state space, must denote a surjective function $f': S_E \rightarrow S_A$, thus specifying the constraints to be non overlapping.

1. The maximal amount of time the transition will use once it is actually started.

In this two level abstraction, the former abstraction is used to decrease the description of the object to something useful at the task level, and the latter to define the semantics of the object and to provide a base for error recovery.

Since the state of an object may now be abstracted to an abstract state for that object - a single variable - the state space used in any planning effort would not increase exponentially with the number of state variables used to control the system, only with the number of objects it contains.

Task level programming - coordinating object activities

The task level language has the role of coordinating object activities. The most primitive way to do this is to request a single object to change the value of one or several of its WM variables. This is a *primitive action*. A *user defined action* is a collection of calls to primitive and/or to user defined actions, and a *partial temporal order* (PTO) over them. The PTO consists of a set of *temporal restrictions* over pairs of these action calls. A temporal restriction between two action calls specifies that they must execute in sequence and which of them that should execute first. When executing a user defined action, the default is to execute all its parts in parallel, except when the PTO prohibits this. Whether parallel actions actually will be executed in parallel or not, depends on resource management. The control level may not be powerful enough to execute all requested algorithms in parallel, or some object may be occupied by other activities and the requested action will have to wait.

The purpose of using this scheme is to promote the encoding of what ordering restrictions of the application that really exists, and leave to the system to handle further restrictions imposed by resource management.

A set of processes (called *workers*) repeatedly actions to solve their tasks. These workers are activated and deactivated by a human operator, acting as a foreman, deciding what to perform at the shop floor.

The description above is not complete. There exists constructs used within an action to define alternatives, and recursive actions are used to express iterations. There are constructs to express data flow between actions and transformations of data through the use of functions. For a complete description, se [Törne, 1990; Loborg and Törne, 1991].

Error Recovery - exception handling and planning

Examples of commonplace corrective actions in a manufacturing context are to replace what is faulty or free what has accidentally got stuck. However, this may not always be accomplished automatically, and human intervention may leave the machinery in another state than a state appropriate for continued execution.

The following sections will briefly present the exception handling to be used in a layered system as described above, and present some ideas about how to support the user when the state of the machinery no longer corresponds to what is appropriate to continue execution.

Since this support will mainly depend on the object specification, it is expected that many task level languages that resembles the Aramis language as presented above may use a similar aproach.

Execution and error semantics

Based on previous sections, the following definitions are straight forward:

- The *referential closure* of a task level process[1] *p* (RC_p) is the set of all state variables from objects which are referenced by *p*. The idea is to limit the world to be concidered during recovery situations. The *abstract referential clousure* (ARC_p) is the set of objects referenced by *p*.
- A *process state* for a process *p* is a vector of values for the variables in the RC_p. An *abstract process state* is a vector containing the current abstract state for each object in the ARC_p.
- The *process state space* is the set of all possible states of a process, and is divided in three subsets:
  - The *illegal state space* is the subset where some of the objects are in an illegal state
  - the *abnormal state space* is the set of legal process states not normally passed during execution
  - the *normal state space* is the set of legal process states passed during some execution

  Analogous, there is an abstract counterpart.
- An *execution path* for a process *p* denotes any sequence of process states for *p* that may be passed during some execution[2]. All possible execution paths for *p* forms an *execution graph* for *p* (EG_p). A *normal execution path* (NEP_p) denotes the trace of an execution of an errorfree program, i.e. a program which succeeds in accomplishing its task. The subset of the EG_p that is normally used is denoted a *normal execution graph* for *p* (NEG_p). The abstract versions (AEG_p and $ANEG_p$) are defined analogously.
- An *error* occurs when an execution deviates from the EG_p and there is no exception handler that succeeds in redirect the execution back into the EG_p. The error will manifest itself in any of the following three alternatives:

---

1. Actually, only a top level process, a worker. Although action calls inside the worker are performed in parallel and thus constitutes processes, they are regarded as action calls - not processes.

2. With the term ‘some execution’ we do not include execution of exception handlers!

- the execution enters the illegal state space
- the execution halts prematurely
- explicit monitoring of the process state detects the deviation

This definition is based on the assumption that the model of the world is correct, but not complete (thus the second point above).

- An *exception* occurs when an execution deviates from the EG_p but is redirected into the EG_p (without entering the illegal state space) by some exception handler.

As a result of this definition, problems during the assembly process caused by damaged parts or parts slightly out of specification are not regarded as errors if there exists a procedure to recover from these situations.

Generally, recovering from an error corresponds to changing the state of the machinery so it will correspond to any process state in the EG. It is preferable to recover to a process state in EG 'close' to the current process state - provided it is possible to associate a cost function to the transitions of objects. Using this approach, the following issues must be concidered:

- deciding whether a process state is in the EG or not is non trivial, compared with deciding whether the state is inside a NEG - a matter of comparing with the execution log.
- simply changing the state of the machinery will most likely conflict with physical or other restrictions on how to operate the machinery. The information about such restrictions is encoded in the task level program, explicitly or implicitly. That information must be extracted to guide the planning of how to change the state of the machinery.

The following subsections will present suggestions on how to implement exception handling and a restricted version of recovery from errors, based on the definitions and discussion above.

Exception handling in the control level

From the task level point of view, it is preferable if the control level could behave in a transactional way, i.e. either accomplish requested state changes or report a failure without changing anything. This is not always possible - a half done seam weld may not easily be undone - but should be desired whenever possible.

A straight forward approach is to label transitions of an object as undoable and/or restartable, either by specifying other algorithms to use in order to achieve that, or by specifying the object to use another existent transition and its algorithm as undo or recover transition. This is analogous to the notion of 'compensating actions' and has been proposed in manufacturing applications by, e.g. [Schmidt, 1992]. A further functionality is to specify the transition as interruptable, i.e. specifying that the task level is to execute some prepared but exceptional task before returning the control back to the control level.

The definition of objects as proposed herein is easily adapted to such a solution, but experiments with different applications is needed to evaluate this approach.

Exception handling at the task level

Providing exception handling at the task level may be viewed as simply providing some syntactic construct to catch a signal and transfer control to some predefined piece of code. Examples of such exception handling constructs can be found in several different programming languages. Most of them have in common that they describe how to handle exceptions in a sequential set of instructions - not parallel activities nor higher level instructions/activities.

In the data base community, the notion of SAGAS or nested SAGAS [Garcia-Molina and Salem, 1987; Garcia-Molina *et. al.*, 1991] has been developed and proposed for modelling parallel, nested activities in a corporation, such as receiving orders, billing the customer while updating the inventory, and so on. In principle, it is a scheme for specifying compensating activities to be used in case of an abortion of an ongoing activity. It also specifies how the abortion of one activity that is a part of a nested structure of activities should affect the other activities. There are no means to describe that an alternative activity should be performed when an activity aborts.

In a more general setting, as described in previous sections, it might also be valuable to be able to specify additional and/or alternative actions to use, as well as how to return to normal execution. The latter is uncommon in normal programming languages such as ADA, where the exception handler 'takes over' the remaining execution in the module that experienced the error. If the exception handler succeeds, the module will return as normal, otherwise it will propagate the same or a modified error to its caller.

Error Recovery at the task level

When the exception handling fails, the error is propagated up to the user, the human operator. He or she has to understand the original fault and correct it in order to continue execution. In order to correct the fault, it is not uncommon that the state of the machinery is changed merely in order to access the failing device, nor is it uncommon to remove some of the work pieces due to a defect - either causing the problem or a result of it. The effect of this activity is, that the machinery is not always in the same state as it was when interrupted. As specified previously, the process state after the failed action may now be either illegal, abnormal or normal.

Using the current definition of an object there is no support for how to change an illegal process state into a legal one, i.e. the system can not propose how to do it. It can merely present the fact and inform the user about the expected state.

If the process state is abnormal, it indicates that no execution passed this state yet. Thus the state must be changed to be a normal one in order to be certain that it is possible to continue execution. If this normal state does not correspond to the one where the fault occurred, it is likely that continued execution will fail. The reason is that the internal state of the controlling computer is not correct with respect to continued execution from this new, normal state.

The execution log - a source of information It is a basic assumption in this presentation that the information needed to perform recovery planning is either explicitly available or may be extracted by observing the system executing under normal conditions. This section will discuss the second part of that assumption.

The referential closure of a process p (RC_p) is an interesting notion since most problems tend to be local to the process[1]. A straight forward method to identify a RC_p used in an application is to log the activities of process p. The RC_p may also be analyzed from the source code for p.

As stated previously, the normal execution graph (NEG) is useful for deciding whether a legal process state is normal or abnormal, i.e. if there exists a program which can be used to continue execution from that process state. By logging the process state the NEG (and the ANEG) is obtained. Since logging is *process local*, the log will start to repeat itself, i.e. as processes continue to execute cyclically, the log of each process will contain repeating series of log records. When that happens, new log records will no longer be appended to the log. The existing log will be reused, expanded with branches if the path starts to deviate from the previously known cycle. Extra care must be taken when handling variables referring to material in flow, since they may point to different object instances at different points in time.

Apart from logging the state of the system, the state of the controller must be logged in order to support resynchronization of program execution with a state of the machinery.

The result of this logging scheme is a knowledge base with information about the normal behaviour of the application, i.e. knowledge about data and control flow. Thus implicit knowledge is extracted and made explicitly available for reuse.

Planning for error recovery Planning is needed in order to support the user in situations when the machinery is in an abnormal state. Rather than creating a plan from the current (abnormal) state to the last known normal state, we suggests that a plan should be generated to the closest normal state. Normal execution is then restarted from that new state, using the logged controller information to synchronize the controller with the state.

The expected advantage of using this approach, is that it will exclude the need to plan for a repetitive sequence of actions - thus reducing planning complexity. The motivation is postponed to the end of this subsection.

The cost function implied by the term *closest* used above is defined as follows:

- for each object there is a cost associated with each transition between abstract states of that object
- given a current abstract state for an object, the distance two states in that object is defined as the minimal sum of transition costs between the states. Since transitions have a direction, the distance between state A and B of an object may differ from the distance between B and A. The distance is infinite if there is no sequence of transitions from A to B.
- the distance between two abstract process states is the sum of the distance for the components, i.e. the individual objects.

Using this definition the distance between to process states is defined, using the abstract state space, as the shortest distance between the corresponding abstract process states.

Producing a plan to reach the selected goal state amounts to selecting the order in which the different transitions from the participating objects should be executed. If no ordering could be found, there are physical restrictions in the machinery that prohibits the current solution. In such a case the next cheapest goal state is selected and the process restarts[2]. When the ordering is found, the plan is refined from the abstract state space to the full state space as represented in the world model. This refinement is safe, i.e. it will not invalidate the plan achieved at the abstract level.

Normally, deciding when it is safe to use a planning operator implies verifying pre- and prevail-conditions for that operator. In this approach, each transition can be viewed as an operator. By analyzing the log, pre- and post-conditions as well as a pessimistic version of the prevail condition can be extracted for each transition. Since the planning is performed in the abstract state space, these conditions will express the current abstract states of other objects in the RC_p, or a “don’t-care” value.

This approach does not permit any object to re-enter a state it has already visited. This restriction is natural - if the application implies a partly repetitive behaviour, i.e. that an object cyclically returns to the same state, there will also exist programs used during normal execution to accomplish this behaviour. Thus these programs may be used from whatever state in the loop the planner choose as goal state.

1. The case when process synchronization is a part of the problem is postponed as future work.

2. In order to include an alternative, more expensive set of transitions leading to the same goal state, the definition of distance must be rephrased.

In [Klein, 1993; Bäckström, 1992; Bäckström and Klein, 1991] a planing formalism called *simplified action structures* (SAS) and an extended version of it (SAS^+) is presented. Using this formalism, some restricted planning problems is shown to be tractable, i.e. solvable in polynomial time. The approach presented herein is close to but does not fully correspond to these restricted classes. The main advantage with this approach is that the actual planning problems as posed by a real application is expected to be limited, both in terms of the state space (depending on how processes are structured) and in the length of produced plans.

Resynchronization A seldom addressed problem when dealing with an error recovery situation in the automation context is, that even if the human operator is capable of manually modifying the state of the machinery to be a legal one, it is almost impossible to continue execution from that state. The reason is that it is non trivial to change the *internal* state of the controller to correspond to the new state of the machinery. However, if there is a well defined separation between different processes in a controller, and if inter-process communication is formalized, it is possible to support the change of the internal state for one such process.

As described above, logging of a process implies saving the values for the state of the process (all variables in the RC_p) as well as information about the current activation records used by the process (the internal state), i.e. both data and control information. Using this information, it is straight forward to modify the internal state[1] to be in accordance with any state present in the log; Find the entry in the log corresponding to the state, and replace the process internal control information by the one found in that entry.

Conclusion

In this paper, a proposal is presented of how to structure a manufacturing control system in order to achieve an open, flexible and modular system which is capable of reusing the knowledge encoded for execution to provide support for error recovery. The knowledge is of the same type as frequently used in proposals for detection/monitoring and diagnose, thus enabling its reuse for those purposes as well.

References

[Bäckström, 1992] C. Bäckström. Computational Complexity of Reasoning about Plans. PhD-thesis no.281, Linköping University, 1992

[Bäckström and Klein, 1991] C. Bäckström, I. Klein. Planning in polynomial time, the SAS-PUBS class. Computational Intelligence, 7:181-197, August 1991.

1. Provided that the executing system is well defined, using modifiable data structures to hold such information. If the executing code is compiled to machine code, this may not be the case.

[Chen and Trivedi, 1991] C.-X. Chen and M.M. Trivedi. A task planner for sensor-based inspection and manipulation robots. In *Proceedings of the SPIE - The International Society for Optical Engineering*, vol.1571, p. 591-603, 1991

[Cao and Sanderson, 1992] T. Cao and A.C. Sanderson. Sensor-based Error Recovery for Robotic Task Sequences Using Fuzzy Petri Nets. In *Proceedings of IEEE International Conference on Robotics and Automation*, p.1063-9, 1992

[Delchambre and Coupez, 1988] A. Delchambre and D. Coupez. Knowledge based error recovery in robotized assembly. In *Proceedings of the 9th International Conference on Developments in Assembly Automation - Japan vs Europe; Product Design for Assembly; Assembly Automation*. p.349-66. IFS Publications, Kempston, Bedford, UK, 1988

[Fohler, 1992] G. Fohler. Realizing Changes of Operational Modes with Pre Run-Time Scheduled Hard Real-Time Systems. In *Proceedings of the Second International Workshop on Responsive Computer Systems*, Saitama, Japan, October 1992.

[Gaspart *et. al*., 1989] P. Gaspart and A. Delchambre and A. Coupez and P. Brouillard. Rule based procedures for diagnosis and error recovery. In *Proceedings of MIV-89 - International Workshop on Industrial Applications of Machine Intelligence and Vision* (Seiken Symposium). p.88-93, 1989

[Gini, 1983] M. Gini. Recovering from Failures. A New Chalenge for Industrial Robotics. In *Proceedings of the 25'th IEEE Computer Society International Conference (COMPCON-83)*. p.220-227, Arlington 1983

[Garcia-Molina and Salem, 1987] H. Garcia-Molina, K. Salem. SAGAS. Proc. SIGMOD int. conf. on Management of Data, pp.249-259, May 1987

[Garcia-Molina *et. al.*, 1991] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner and K. Salem. Coordinating Multi-Transaction Activities. Data Engineering Bulletine 1991 (also in COMPCOM91)

[Klein, 1993] I. Klein. Automatic Synthesis fo Sequential Control Schemes. PhD-theses no.305, Linköping University, 1993

[Kopetz, 1992] H. Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. *Proc. 12th Int. Conference on Distributed Computing Systems*, Japan, June 1992

[Kopetz and Ochsenreiter, 1987] H. Kopetz, W. Ochsenreiter. *Clocksynchronization in Distributed Real-Time Systems*, IEEE Transactions on Computers, Vol C36, No. 8, pp. 933-940, Aug. 1987

[Lee *et. al.*, 1983] M.H. Lee and D.P.Barnes and N.W. Hardy. Knowledge Based Error Recovery in Industrial Robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*. pp.824-826, 1983

[Leinbaugh and Yamini, 1982] D. W. Leinbaugh and M. Yamini. Guaranteed response time in a distributed hard real-time environment, *Proc. Real-time systems symposium*, Dec. 1982.

[Loborg *et. al.*, 1994] P. Loborg, P. Holmbom, M. Sköld and A. Törne. A Model for the Execution of Task Specifikations for Intelligent and Flexible Manufacturing Systems, accepted for publication in *Integrated Computer-Aided Engineering* (special issue about AI in Manufactoring and Robotics to appear somewhere during 94).

[Loborg et. al., 1993] P. Loborg, T. Risch, M. Sköld and A. Törne. Active OO Databases in Control Applications, in Microprocessing and Microprogramming, Vol. 38, No 1-5, p.255-264. proceedings of the *19th Euromicro Conference*, Barcelona, sept 1993.

[Loborg and Törne, 1991] P. Loborg and A. Törne. A Hybrid Language for the Control of Multimachine Environments, in Proceed*ings of EIA/AIE-91*, Hawaii, June 1991.

[Ma *et. al.*, 1988] P. R. Ma, E. Y. S. Lee and M. Tsuchiya. A Task Allocation Model for Distributed Computing Systems, *Hard Real Time Systems*, Eds. Sytankovich/Ramamritham, pg.249-255, IEEE Comp.Soc.Press, 1988.

[Meijer and Hertzberger, 1988] G. R. Meijer and L. O. Hertzberger. Off-Line Programming of Exception Handling Strategies. In *Proceedings of IFAC Symposium on Robot Control*. p.431-436, Karlsruhe 1988

[Meijer *et. al.*, 1991] G. R. Meijer and L. O. Hertzberger and T. L. Mai and E. Gaussens and F. Arlabosse. Exception Handling System for Autonomous Robots Based on PES. In *Journal of Robotics and Autonomous Systems*. p.197-209, no.7, 1991

[Rajkumar *et. al.*, 1988] R. Rajkumar, L. Sha, J. P. Lehoczky, Real-Time Synchronization protocols for multi-processors. *Proc. IEEE Real-Time systems symp*., CS Press, Los Alamitos, Calif., pp. 259-269, 1988

[Schmidt, 1992] U. Schmidt. A Framework for Automated Error Recovery in FMS. In *Proceedings of the 2'nd International Conference on Automation, Robotics and Computer Vision*. p.IA.3.4.1-5, 1992

[Sha and Sathaye, 1992] L. Sha and S. Sathaye. Distributed Real-Time System Design using Generalized Rate Monotonic theory, *Second Int. Conf. on Automation, Robotics and Computer Vission*, SINGAPORE, 1992

[Srinivas, 1977] S. Srinivas. Error Recovery in Robot Systems PhD thesis California Inst. of Tech., Pasadena, 1977

[Srinivas, 1978] S. Srinivas. Error Recovery in Robots Through Failure Reasoning Analysis. In *Proceedings of AFIP - National Computer Conference*, p.275-282, 1978

[Stankovic *et. al.*, 1990] J. A. Stankovic, K. Ramamritham, and P.-F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 184-194, April 1990

[Taylor and Taylor, 1988] G. E. Taylor and P. M. Taylor. Dynamic error probability vectors: a framework for sensory decision making. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, p.1096-100 vol.2, 1988

[Taylor *et. al*., 1990] P. M. Taylor and I. Halleron and X. K. Song. The application of a dynamic error framework to robotic assembly. *IEEE International Conference on Robotics and Automation*, pp170-5. IEEE Comput. Soc. Press, Los Alamitos, CA, USA 1990.

[Thomesse *et. al.*, 1991] J.-P. Thomesse, P. Lorenz, J. P. Bardinet and T.Valentin, *Factory Instrumentation Protocol: Model, Products and Tools*, Control Engineering, pp65-67, Sept 1991.

[Törne, 1990] A. Törne. The Instruction and Control of Multi-Machine Environments, in *Applications of Artificial Intelligence in Engineering V, vol. 2, pp. 137-152, proc. of the 5th Int. Conf*. in Boston July 90, Springer-Verlag, 1990.

[Verhoosel *et. al*., 1991] J. P. C. Verhoosel, E. J. Luit, D. K. Hammer, E. Jansen. A Static Scheduling Alghorithm for Distributed Real-Time Systems, *Journal of Real-Time Systems*, 3, pp.227-246, Kluwer Acad. Pub. 1991.