VTT PUBLICATIONS 283

A control architecture for intelligent robots based on graph manipulation for planning, monitoring, execution, intention switching, and fault recovery

Jörg-Michael Haseman

VTT Automation

Dissertation for the degree of Doctor of Technology to be presented, with the permission of the Department of Electrical Engineering of the University of Oulu, for public discussion in Auditorium L 10, Linnanmaa, on August 30th, 1996, at 12 o'clock noon.

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 1996

ISBN 951-38-4941-4
ISSN 1235-0621

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Automaatio, Koneautomaatio, Kaitoväylä 1, PL 13023, 90571 OULU
puh. vaihde (981) 551 2111, telekopio (981) 551 2320

VTT Automation, Maskinautomation, Kaitoväylä 1, PB 13023, 90571 ULEÅBORG
tel.växel (981) 551 2111, telefax (981) 551 2320

VTT Automation, Machine Automation, Kaitoväylä 1, P.O.Box 13023, FIN-90571 OULU, Finland
phone internat. +358 81 551 2111, telefax +358 81 551 2320

Technical editing Leena Ukskoski

VTT OFFSETPAINO, ESPOO 1996

Hasemann, Jörg-Michael. A Control architecture for intelligent robots based on graph manipulation for planning, monitoring, execution, intention switching, and fault recovery. Espoo 1996, Technical Research Centre of Finland, VTT Publications 283. 145 p. + app. 5 p.

UDC 681.5:007.52.681.3
Keywords robots, control systems, models, planning, monitors, switching, fault analysis, automation, artificial intelligence, fuzzy logic, computer programming

Abstract

A new robot control architecture is proposed which combines the functions of planning, intention switching, execution, plan validity and plan execution monitoring, and fault recovery. Emphasis is placed on providing a technological basis for building control systems for intelligent robots by modelling the above mentioned functionalities using simple, though powerful, graph manipulation techniques and introducing different types of plan components for monitoring, planning, and execution using an object-oriented design approach.

The proposed architecture is related to existing approaches and results from the research areas of monitoring and planning. An implementation of the architecture is described and results from tests in a simulated environment are reported. A new method for grasp and motion planning based on hierarchical fuzzy logic rule sets is presented and its feasibility is verified by applying it to an industrial paper roll manipulator.

Preface

The work covered in this thesis was carried out during 1992 - 1996 at the Technical Research Centre of Finland (VTT), VTT Automation, Oulu, Finland.

My employer, VTT Automation, has made it possible for me to do this research. I wish to thank Prof. Pentti Vähä, head of VTT Automation - Machine Automation, and Mr. Klaus Känsälä, research group leader Mechatronics, for supporting my research and arranging me the opportunity to write this thesis.

I am greatly indebted to VTT for providing me the chance to present my work at renowned conferences and visit leading research institutions. I am very grateful for the influential comments on my research by Dr. Reid Simmons from the Carnegie Mellon University, Dr. Kurt Konolige from Stanford Research Institute (SRI), Dr. Pat Langley and Prof. Nils Nilsson from Stanford University, Prof. Aaron Sloman from the University of Birmingham, and Dr. Malik Ghallab from Laboratory for the Analysis and Architecture of Systems (LAAS), Toulouse.

I wish to thank Dr. Tapio Heikkilä from VTT Automation for his support and guidance of my research work throughout my 5-year long stay at VTT. I thank Douglas Foxvog for proofreading the thesis. Remaining mistakes are my own.

I further like to thank Prof. Olli Silvén from Oulu University for supervising this thesis and Prof. Masayoshi Kakikura from Tokyo Denki University and Prof. Juha Röning from Oulu University for reviewing this thesis.

Finally, I wish to thank my wife Sari for her patience and understanding during the time of writing this thesis.

Oulu, June 1996

Jörg-Michael Hasemann

CONTENTS

1 INTRODUCTION

This thesis is about robot control architectures, the requirements for robot control architectures, the problems arising when creating and designing control architectures for intelligent robots, and ways to advance beyond current practices.

We envisage intelligent robots as a technological advancement from today's human-operated service and utility machines such as vacuum cleaners, lawn movers, straddle carriers, and forklifts. The next generation of intelligent robots, so called service robots, will push automation into fields of human services previously unexposed to automation. Such areas include courier robots for delivering of office mail or cleaning robots for cleaning, e.g. offices, public places, and sanitary installations.

One application scenario of intelligent robots is an automated harbour environment in which paper rolls arrive by train, are unloaded, and transported to a waiting ship. In this scenario, autonomously guided vehicles (AGVs) and paper roll manipulators (PRMs) are the robots to be controlled by control systems which follow the proposed robot control architecture. Although this scenario is in many respects oversimplified it serves well to elaborate the problems and difficulties robot control architectures are faced with.

Robot control architectures are concerned with organising perception and action in order to achieve given goals. Goals can be categorised as achievement goals, e.g. "unload the ship and transport all paper rolls to the store", maintenance goals, e.g. "make sure that the AGV's fuel level is sufficient", and avoidance goals, e.g. "do not drive into other objects". Perception provides the control system with information about the environment, which enables it to plan for its goals.

Planning means deciding what to do next in order to achieve a given goal. Planning can be carried out either for a very limited planning horizon, i.e. planning just the immediately following activity, or long term, i.e. planning far ahead. Planning far ahead offers a possibility to optimise activities and increase efficiency. Long term planning and the maintenance of long term plans are costly and vulnerable to unexpected events. In our example scenario unexpected events can have very different origins:

- Vehicles or humans may interfere, cross, or block a robot's planned trajectory causing it to replan its path to get the paper roll to the final target place.

- Greater than expected fuel consumption may cause the robot to refuel earlier than planned.

- Perception of the environment might be failure prone. Sensors may fail or report incorrect information.
- Actuators, motors, communciation, etc. may fail or break down.
- Planned activities may fail.

These events cause deviations from the normal, i.e. planned, flow of activities and demand replanning in order to recover from surprise situations. Due to the far horizon of long term plans they are more vulnerable to unexpected events than short term plans. Moreover, recovery within complex plans is much more difficult due to the higher number of dependencies among different activities within a plan.

In order to address the problem of unexpected events, they first must be detected. Monitoring performs this activity. All conditions possibly affecting the success of plan execution must be monitored in timely fashion to guarantee success. For example:

- When transporting a paper roll from one place to another, the planned path must be surveyed to detect objects blocking the way.
- To avoid running out of fuel, the fuel level must be monitored.
- To detect sensor errors, the plausibility of sensor readings should be checked.
- Actuators and other devices must be monitored to validate that they behave as planned.

Monitoring, not only implies validating of plan execution but, perhaps as important, validating the correctness of planned activities with respect to changes in the environment. A paper roll manipulator, after taking a paper roll from a train car, may find the planned placement location in the buffer area occupied by another paper roll or another more appropriate site might become free that moment.

Obviously, the planned activity is either not valid anymore, because the target site is not free, or is unnecessarily inefficient, because a better target site could be used. Detecting unexpected events, i.e. changes in the environment which (may) affect plan execution either positively (serendipitous event) or negatively (adverse event) at an early stage, enables early replanning and thus increases efficiency. Hence, an unexpected event may affect plan execution both positively or negatively, although experience teaches that unexpected events are usually negative.

We do not yet have intelligent robots vacuum cleaning our living rooms, servicing drinks and pizza late night, dispatching office mail at our work places, assisting handicapped people in their daily life, or accomplishing automation tasks such as loading and unloading train cars or ships, although the major problems are seemingly solved: powerful and fast algorithms exist for performing crucial tasks in intelligent robots. Such well established techniques include traditional control, fuzzy control, fuzzy decision making, neural networks, symbolic domain-dependent planning of hierarchical task networks, and path, trajectory, and motion planning.

Moreover, developments in material sciences and sensor technologies bring us cheaper, lighter, and more reliable sensors, actuators, batteries, and structural components.

At the same time, the industrialised world is undergoing changes towards service sector economies with high demands for affordable services and an (expected, though in the current economic situation unbelievable) increasing unavailability of low wage personnel, i.e. with wages at a level which allow services to be commercially viable.

Increasing environmental awareness results and will result in regulations enforcing the introduction of intelligent robots for traditional tasks that are or will then be considered hazardous to workers or new tasks, such as recycling, waste disposal, environmental inspection, or surveillance tasks. All this suggests, that we should already experience the dawn of intelligent robots for service tasks, but we do not.

We claim that the major obstacle of turning robots for service tasks into reality is the lack of an architectural framework integrating and coordinating logically or physically distributed activities such as planning, sensing, monitoring, and acting.

1.1 THE SCOPE OF THE THESIS

The purpose of this thesis is to develop concepts and techniques for robot control architectures. Particular emphasis, apart from architectural and integration issues, is given to aspects of plan execution and plan validity monitoring, intention switching, and problems of action representation.

Robot control architectures provide concepts for how to organise logically distinct activities in order to achieve intelligently behaving robots, with timely and appropriate response to external triggers. Unreliable sensors and actuators, incomplete and partially incorrect models and beliefs about the environment, as well as the requirement to react to sudden external or internal events render these tasks extremely difficult despite the fact that well researched methods and algorithms for various subtasks such as planning, sensing, servo control, and obstacle avoidance exist. One major reason for this is the still open question of how to organise different subtasks within a robot control architecture.

The scope of this thesis is, hence, to propose methods to address the crucial tasks of planning, execution, monitoring of plan execution and validity, and intention switching, and their arrangement within a robot control architecture.

- **Planning** denotes action selection and action arrangement on various levels of abstraction, ranging from sophisticated high-level deliberative feedforward decision making such as operator arrangement, resource consumption and allocation, and risk assessment, to low-level reactive and reflexive spontaneous action selection.

- **Execution** denotes the timely execution of physical and non-physical activities such as actuator commands and sensing procedures.

- **Monitoring** denotes monitoring execution of actions and the validity and consistency of planned activities with respect to ongoing changes in the environment.

- **Intention switching** denotes the existence of conceptually distinct tasks, such as recharging batteries or transporting goods. These tasks are continuously evaluated and their criticality is determined. Depending on their criticality, tasks may be initiated and, if necessary, less critical ones interrupted and later resumed.

The objectives of this thesis are to devise a representation of plans consisting of constraints, actions and ordering relations, i.e. hierarchies, sequence, and parallelism, as well as methods to react to asynchronous events, such as faults and triggers, which demand fast attention and reaction.

The proposed robot control architecture was designed for intelligent robots with medium task complexity operating in a dynamic environment. Service robots fall into this category as they typically have a low to medium task complexity with respect to the amount and sophistication of scheduling and autonomy required. Their tasks such as cleaning, navigation, or manipulation, typically have far reaching goals which can only be achieved by execution of many operations. For example, delivering office mail from one location to another involves many different activities such as planning paths, navigation, following corridors, and opening doors. Most often such tasks can be well described by an expert and can be based on (nested) task net templates in the design phase.

Typical applications of intelligent robots for service tasks put often little emphasis on scheduling and detailed long term planning but demand reactivity and robustness due to a high likelihood of unexpected events in their environments, which are frequently present intelligent robots with unpredictable situations such as blocked or locked doors, obstacles, and interrupting high priority tasks. For these reasons, template based planning, fault detection by monitoring plan validity and execution, and fault recovery are the central issues covered by this thesis.

Some of the concepts in this thesis have been developed for and used within a large scale paper roll manipulator designed to automate loading and unloading of ships and train cars in harbour areas. Illustrative examples as well as results from implementations provide illustration and evidence of the usefulness of the proposed techniques.

1.2 THESIS MAP

Because this thesis emphasises technological issues concerning the design, development, and implementation of a robot control architecture for intelligent robots, the thesis is organised as follows.

In Chapter 2, “Problem Areas”, we start with a review of the theoretical background of planning and control, relate them to each other, and discuss complexity results for various types of classical artificial intelligence planning. In order to cover the whole spectrum of planning, a short discussion on low-level planning and control follows to close this section.

The second part of this chapter is a discussion of monitoring. We relate monitoring to diagnosis and recovery, distinguish between monitoring discrete and continuous event systems, and review a number of proposed approaches to monitoring for robot control architectures. Chapter 2 closes with a discussion of intention switching and fault recovery.

Chapter 3, “Robot Control Architectures”, discusses current approaches to robot control architecture and compares a number of influential ones according to the dimensions and attributes of their applications.

In Chapter 4, “The Architecture”, we propose a new robot control architecture, which combines important features, such as concurrency, planning by decomposition, task switching, task initiation, object-oriented layout using plan component classes, methods for fault recovery, plan execution and plan validity monitoring. As the central chapter of this thesis, the theoretical framework of the implementation described in Chapter 5 is built here.

At the end of Chapter 4, we briefly review the results achieved with JANUS, a monitoring shell developed for a paper roll manipulator, which incorporates methods to derive postcondition-precondition relationships from STRIPS-based plan structures to be used for monitoring purposes.

In Chapter 5, “Experiments and Results” we describe an implementation of the robot control architecture and present results gained from simulations. The implementation covers all major concepts introduced in the previous chapter and includes test runs carried out in a simulated material flow environment with one autonomous guided vehicle and one autonomous paper roll manipulator. Although the test runs have been carried out in a simulated environment only, they still demonstrate the usefulness of this approach for a broad class of robotic tasks with high - but not full - autonomy, low requirements on scheduling, but a high likelihood of disturbances and interferences during operation.

In order to make the implementation as complete as possible and to elaborate on the interactions between the symbolic layer and lower non-symbolic layers, we also describe one non-symbolic low level control task. This control task concerns complex motion and grasp planning and is accomplished by hierarchical fuzzy modelling. This part of the implementation, together with the fuzzy logic interpreter and the geometric reasoner, is used in a real intelligent robot, i.e. a 16 ton industrial paper roll manipulator.

Chapter 6, “Conclusions” closes this thesis with comments on the achieved results. The appendix lists all definitions used in this thesis to provide an easily accessible reference.

1.3 THE CONTRIBUTIONS OF THE THESIS

This thesis proposes new concepts for the design of control systems for intelligent robotic systems for which expert knowledge of how to accomplish the robot’s tasks can be obtained and formulated as constrained plan templates. Such systems range from from teleoperated manipulators to autonomous mobile robots for delivery and transportation tasks and cover the popular field of service robotics.

The concentration on such systems is application motivated by the expected demand for automating *complex* but *defined* tasks. For the same reason we did not investigate learning aspects which under different circumstances, e.g. high degree of autonomy and unclear task decomposition, pose an important issue for robot control architectures.

Possible applications of the proposed robot control architecture are robotic tasks typically with a high likelihood of disturbances and interferences during operation and high but not total autonomy. The proposed robot control architecture, however, is not deemed to be suitable for tasks with a low likelihood of disturbances and interferences, a more or less static work environments, and high demands on scheduling, such as large batch number production and manufacturing scenarios, which are more appropriately addressed by traditional off-line schedulers and production planning systems.

The key contributions of this thesis to the field of intelligent robotics lie in

- the representation of intentions within the current execution graph. Intentions describe what the robot is going to do next. Although the use of graphs to represent and visualise plans is common, the current execution graph acts as a dynamic visual cue for system developers and robot operators to show the *progress* of planning and execution and the *effects* of intention switching and fault recovery are new.

- the manipulation of the current execution graph as a result of planning and execution. The proposed robot control architecture owes much of its flexibility to the formulation of its functionalities as graph manipulations (for planning, execution, fault recovery, and intention switching) and operations on graphs (plan validity and execution monitoring). Many classical AI planning methods, particularly hierarchical task network planning, can be easily represented as graph manipulations. The particular innovation of the proposed architecture lies in augmenting graph manipulation for hierarchical task network planning with additional graph manipulations for other essential and imperative functionalities of robot control architectures such as execution, fault recovery, intention switching, and monitoring.

- the explicit representation of constraints within plan structures for plan execution monitoring, plan validity monitoring, and plan merging. Apart from abstract actions, which are further decomposed, and atomic actions, which are tied to control loops, constraints are explicitly represented. This is a new approach to robot control architecture. Explicit constraints within the plan structure are advantageous for several reasons: they can (a) be directly tied to execution monitoring tasks, (b) constrain further task decomposition, and (c) constrain intention switching. Thus, they have both a descriptive and a declarative character. Constraints share the functionalities of pre- and postconditions (add-, delete-lists) in STRIPS-like operators as well as causal links (precondition-postcondition relationships) in STRIPS-like plans. Moreover, constraints are used to watch transient conditions during plan component execution. Transient conditions describe expected partial world states during execution. Such conditions can result from execution of a the related physical plan component or describe a safety envelope of world states under which execution is permitted in order to guarantee safe execution for either the robot or the work environment.

  As an example of a low level behaviour and its integration within the proposed architecture we designed and implemented a path and grasp planning behaviour based on a new method of hierarchical fuzzy modelling which in itself presents a new and computationally inexpensive approach to various path and motion planning problems. The planning behaviour for path and motion planning is currently used within the control system of a 16 ton paper roll manipulator. Its usefulness has been shown in demonstration runs in harbour settings.

- the formulation of fault recovery and intention switching in order to accommodate spurious high criticality tasks. Intention switching is analogous to task switching in operating systems and denotes context dependent engagement and disengagement of high level behaviours. Intention switching is an important and desirable functionality of robot control architectures that has not yet been recognised as such. It enables the implementation of high level behaviours which implement and correspond to conceptually independent tasks such as vacuum cleaning and battery recharging. Similar to the treatment of concurrent tasks by operating systems, the proposed robot control architecture determines task switching points and takes care of the task switching. Intention switching and high level behaviours constitute an important contribution to the field of robotics as they allow a system designer to concentrate on the implementation of high level behaviours rather than on their possible interactions.

The integration of the above into a robot control architecture and the experimental validation of the concepts in a simulated environment demonstrated the expected behaviours and the usefulness of the proposed robot control architecture. Insights are given on how to implement the proposed robot control architecture and how the proposed architecture strongly draws advantages from object-oriented design.

The proposed hierarchical structure of plan templates on the one side and the taxonomy of virtual and physical plan components easily maps onto a hierarchy of classes within the object-oriented paradigm. The use of plan templates as abstract plan components has tool box character and enables reuse for different applications. Although these points might be considered as mere implementational issues, easy implemention and reusability are major cost issues for the broad introduction of intelligent robots.

2 PROBLEM AREAS

Robot control architectures specify how to accomplish and integrate planning, monitoring, and control into a system constrained by (a) imperfect sensors and actuators, which cause failures and misperceptions; (b) restricted computational resources, which limit the amount of planning and data processing; and (c) asynchronous events, such as failures or errors, which call for immediate attention.

The theoretical background for this lies in the fields of planning and monitoring. In this chapter we survey these fields with a particular interest in the computational economy of planning.

2.1 PLANNING AND CONTROL

Within this thesis no conceptual distinction is made between spontaneous action selection (control) and long time look ahead operator arrangements (planning), although the two activities are different in many respects:

- time scale, look ahead, and desired response times
- the nature of underlying information ranging from purely numerical sensor readouts to highly assimilated symbolic knowledge
- the methods used, ranging from classical AI planning, heuristic expert systems, and hybrid planning to PID-controllers, intelligent control approaches such as fuzzy control and neural network control, and reactive subsumption approaches.

Nevertheless, planning and control are just the extremes in a continuous spectrum of acting involving varying degrees of deliberation. Planning and control describe a mapping from initial states to goal states. With increasing abstraction information associated with planning and control, such as state descriptions, goals, and timing constraints, become more and more discrete. Consequently, the mapping becomes a more and more non-linear and non-continuous one.

Traditionally, execution within robot control architecture is implemented by enable signals to hardware drivers perhaps together with control parameters. By including large parts of the hardware drivers' functionalities within the scope of the robot control architecture the former sharp distinction between planning and execution becomes blurred. Including low-level feedback and action loops to the robot control architecture adds hard real-time aspects to the execution of plan components, to be addressed by a designated scheduler.

Depending on the type of application at hand, emphasis on the types of planning and control may vary widely. A teleoperated manipulator obviously has strong demands on sophisticated low-level control tasks such as coordinated movement and force, position, and velocity controlled motions; Autonomous guided vehicles in manufacturing environments may have high demands for elaborate planning and efficient resource usage.

In this chapter, the distinction between low-level planning and control, and "AI-planning" or high-level planning is very pragmatic. It classifies all more or less contemporary high-level approaches as "AI-planning" and everything else (which turn out to be more "low-level") as "Low-Level Planning and Control".

This section starts off with a subsection on AI-planning and describes different approaches within the AI community and sheds light on complexity issues. The second subsection discusses low-level planning and control.

2.1.1 AI-Planning

Traditionally, "AI-Planning" denotes methods which are off-line, i.e. not intertwined with execution, and which produce plans for given situations and goals. Goals, situations, and actions are based on symbolic (propositional) descriptions.

Planning includes ordering of actions, structural decomposition, and instantiation of plan variables. Structural decomposition denotes the mechanism of decomposing complex actions into partially ordered sets of simpler actions. Instantiation is the process of grounding planning variables, e.g. linking the plan variables A,B,C in an action "move A from B to C" to certain objects or places in the real world. Planning proceeds until a precedence graph of fully instantiated ground (atomic) actions has been derived.

This is different from the way people plan. Human plans are incompletely instantiated before execution starts and are completed as detailed information about the environment unveils. This is called intertwined planning and execution and is an important concept of modern robot control architecture. Intertwined planning and execution, however, is prone to errors if expected information does not unveil during execution.

Within the AI community, different algorithms have been proposed for the general planning problem (no time constraints) which can be classified as "from first principle" planners, e.g. precondition achievement planners, which are able to create a correct plan (with respect to the propositional problem description) if one exist (even if it takes millions of years), or heuristic planners which are not guaranteed to create a plan even if one exists, since they only have limited schematic domain-restricted knowledge and search capabilities. Heuristic (e.g. HTN) planners use rich precompiled "procedural human planning knowledge" and produce complex plans by simple instantiation methods.

Correctness and completeness criteria can be fulfilled if these terms are viewed with respect to the given planning knowledge, though time and memory constraints render any method for planning real world problems incomplete.

Ironically, planning algorithms can be classified as provably correct and complete domain-independent ones, i.e. precondition achievement planners with propositional operator and situation descriptions, or practically useful, i.e. domain specific heuristic planners with rich domain-specific knowledge and knowledge-based plan component expansion schemes (Drummond 1994).

Precondition achievement planning (Drummond 1994) in its simplest form denotes planning based on arranging plan components in a way that their pre- and postconditions match and that the initial state is incrementally transformed into the desired goal state by executing the plan. Modern planning algorithms may include more sophisticated methods for searching for correct plans within the plan space, i.e. the set of all possible plan component arrangements (plans). The basic property of all precondition achievement planners is search by successively modifying a partial plan, concentrating on a single unestablished precondition at a time.

The fundamental drawback of all precondition achievement planners is the low expressiveness of operator descriptions and the complexity of the planning algorithm itself which causes huge search spaces even for very simple problems. A simple blocks world example illustrates this search space explosion:

> "Suppose a typical plan for the block world has on average four outstanding goals (hardly an excessive number). Also, suppose that there are on average three ways to achieve each of these goals (very plausible, with the possibility of binding variables, ordering operators, and introducing new operator schemata). This gives us, on average, twelve ways to change an arbitrary block world plan into another one. Each change is designed to remove (at least) one goal (and perhaps introduce others). For an average block world problem, suppose that seven plan modifications are required to change an initially provided plan into one which has no outstanding goals. This tells us that breadth-first search will explore (at worst) 12^7 partial plans. For a seemingly trivial block world domain, a search

space of this size is remarkable.” *(Currie & Tate 1991, p. 55).*

All planning methods discussed in this thesis are inherently situation based, i.e., using situations explicitly to search for correct plans or implicitly by refining hierarchical skeletal plans with prespecified instantiation schemes. For this reason the representation of situations and plan components is discussed in detail. Problems related to the representation of plan components and situations as well as the complexity of planning methods are also reviewed.

In Subsection 2.1.1.2, the theoretical background for planning and monitoring is briefly reviewed. This review is restricted to classical, model-based artificial intelligence (AI) methods and can elaborate only a few aspects of the very complex research area of planning. A more complete survey is given in (Hendler et al. 1990).

2.1.1.1 Representational Problems

All planners are exposed to problems of representing actions and effects. Precondition achievement planners reason about the effects of plan components at planning time and therefore need to have plan component descriptions that closely match the applicability (preconditions) and effects (postconditions) of the corresponding action to produce correct plans.

For the purpose of monitoring also heuristic planners, i.e. those which use plan templates for operator abstraction, need to have a proper plan component description to enable thorough plan execution and plan validity monitoring.

Describing plan components is inherently problematic as a result of the so called ***frame problem***. The frame problem emerges from the difficulty of completely describing what of the world *does not change* as a consequence of the execution of a plan component. Possible solutions to the frame problem are the STRIPS assumptions (coded into the planning algorithm) or a “law of inertia” added as an axiom into the underlying logic or as part of the domain theory. In any solution it is assumed that unspecified changes do not occur, i.e. a plan component has no applicable effects other than those specified.

Related to the frame problem is the ***ramification problem***, which denotes the problem of exhaustively describing *all effects* of the execution of an action under all possible situations. A possible solution is to specify consistent world states as a domain theory.

The ***qualification problem*** is the problem of enumerating *all necessary preconditions* of a plan component. Two cases are distinguished (Schneeberger 1993): (a) something yet unknown prevents successful plan component execution and (b) there are limited computational resources to check all preconditions. Similarly, the ***extended prediction problem*** is the problem that infinitely many events exist in a dynamic world which may terminate a post-condition after its establishment.

2.1.1.2 Traditional General Purpose Planning Systems - STRIPS

The planning systems proposed during the last two decades gradually improved in their performance and the expressiveness of the produced plans. One of the earliest planning procedures was STRIPS ("Stanford Research Institute Problem Solver"; Fikes & Nilsson 1971).

STRIPS uses means-ends analysis to solve problems by successively reducing the difference between the current state and the goal state to guide the search in the operator space. The plan component ("operators" within STRIPS) description used by STRIPS, consisting of a list of preconditions, an add-list (conditions which hold valid as a result of the plan component's execution), and a delete-list (conditions which no longer hold as a results of the plan component's execution), solved the frame problem by assuming (STRIPS assumption) that any condition not specified by the operator remains unaffected during its execution.

STRIPS produces totally ordered (linear) plans by forward planning, i.e., by starting from the initial situation and computing the subsequent situation after virtual application of a plan component. This step is called *progression* within the context of planning. However, backward search, *regression*, or a combination of both plan modification methods (opportunistic planning) may be also applied.

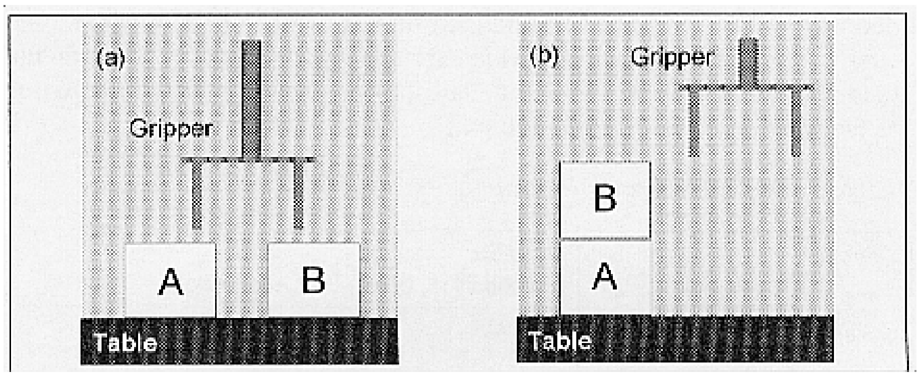


Figure 1. Initial (left) and goal situation (right) of the STRIPS planning example.

Table 1. Predicates and their Meaning.

Property:	***Meaning:***
P_1: ONTABLE(A)	Block A is on the table.
P_2: ONTABLE(B)	Block B is on the table.
P_3: HOLDING(NIL)	The gripper is empty (holds nothing).
P_4: CLEAR(A)	Block A is clear.
P_5: CLEAR(B)	Block B is clear.
P_6: ON(A,B)	Block B is on top of block A.
P_7: HOLDING(A)	The gripper holds block A.
P_8: HOLDING(B)	The gipper holds B.
P_9: ON(B,A)	Block A is on top of block B.

Figure illustrates a simple blocks world scenario consisting of 2 blocks. The scenario is formally defined with equations. Initial and goal situations are described as conjunctions of predicates whose meaning with respect to the example is given in Table 1.

The initial situation (Figureleft) is described by

$$P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5 \qquad (1)$$

The goal situation (Figure right) is described by

$$P1 \wedge P3 \wedge P5 \wedge P6 \qquad (2)$$

The “problem” is to find a plan which transfers the initial situation into the goal situation by arranging two operators, pickup(B) to pickup block B with the gripper and stack(B,A) to put block B on top of block A. Operators are defined by their preconditions and their effects, which are defined by a delete-list, i.e. a list of predicates which after execution of the operator do not hold any longer, and an add-list, i.e. a list of predicates, which become true as the result of the execution (Table 2).

Table 2. STRIPS Operator Description.

Operator:	***Definition*** **(Precondition, Delete-List, Add-List):**
pickup(B)	P, D: $P_2 \wedge P_3 \wedge P_5$
	A: P_7
stack(B,A)	P, D: $P_7 \wedge P_4$
	A: $P_6 \wedge P_5 \wedge P_3$

Obviously, pickup(B) and then stack(B,A) will achieve the goal. Figure shows how the initial situation is transferred into the goal situation.

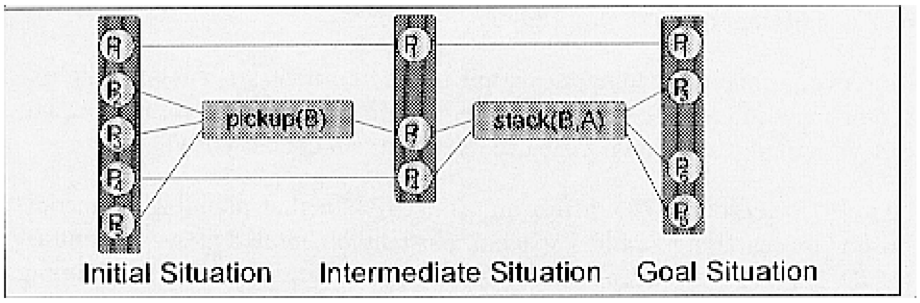


Figure 2. STRIPS Plan.

2.1.1.3 Non-linear General Purpose Planners -NOAH

The next steps following STRIPS were planning procedures which are able to produce partially ordered (non-linear) plans and planning procedures, and which employed situation abstraction to speed up search. Planners employing situation abstraction are also known as hierarchical planners. Examples for hierarchical planners are ABSTRIPS (Sacerdoti 1974) and ABTWEAK (Yang & Tenenberg 1990). ABSTRIPS is based on STRIPS and guided search by a hierarchy of abstraction spaces.

The concept of hierarchies of abstraction spaces is based on the idea that some preconditions are more important than others. Consequently, the planning procedure tries to satisfy the most important preconditions first. The importance (“criticality” in ABSTRIPS) changes whenever the planning procedure fails to produce a plan, e.g. preconditions which could not be satisfied in an earlier planning attempt move higher in the abstraction hierarchy, so that in the following planning rounds these preconditions are tried to be satisfied in an earlier stage. Thus, ABSTRIPS is able to solve problems with generally much less searching and backtracking than STRIPS does.

ABTWEAK, in a similar fashion, adds a hierarchy of abstraction spaces to TWEAK (Chapman 1987). TWEAK is a planning procedure that produces partially ordered plans and is based on ideas partially implemented in NOAH (Sacerdoti 1974).

Partially ordered planners are less committed to ordering the plan components before necessary and as a result may leave many plan components unordered in the resulting plan. Leaving plan components unordered makes planning more efficient, at least for simple plan component representations such as STRIPS-operators. Goldman and Boddy (1994) provide a detailed discussion of the performance of linear and non-linear planners using complex conditional operator descriptions.

Moreover, since plan components are unordered unless necessary, the executor may choose the sequence of execution based on real-time data and (under certain restrictions) may also execute them concurrently.

NOAH (Sacerdoti 1974) differs from STRIPS in that planning is accomplished by searching within a space of partially elaborated plans. In contrast to NOAH, STRIPS searches within a space of situations. NOAH's planning procedure is sketched in Figure 1 (Schneeberger 1993).

1) Initialisation of the plan with an operator taking the initial situation as a precondition and the goal situation as a postcondition.
2) If there are unsolved goals left, place an appropriate plan component in front of the goal. Do not imply an ordering with respect to other plan components.
3) Check whether the unordered plan components' postconditions cause conflicts. If so, try to resolve conflicts by:
   - ⇒ ordering conflicting plan components
   - ⇒ constraining the value of a variable
   - ⇒ add new plan component(s) into the plan
4) If there are still unsolved subgoals left, continue with step 2.

Figure 1. NOAH Planning Procedure.

Step 3 in Figure 1 contains the QA Algorithm (Question-Answering; Sacerdoti 1974) which determines whether a proposition has a specified value at a certain point in a partially ordered network. This QA algorithm was later formalised as the Modal Truth Criterion (Chapman 1987) indicating whether a partially ordered plan is correct or not and is now part of all modern AI planning systems (according to Tate 1994).

NOAH has been used as a support system to assist technicians in repairing technical devices. Other planning system used outside block worlds include NonLin (Tate 1977), SIPE (Wilkins 1983), and O-Plan (Tate 1994). Their success, ac cording to Drummond (1994) lies partly in hierarchical plan component expansion (knowledge based planning by plan component expansion instead of search based planning), explicit languages for documenting a plan’s causal structure, and a very simple form of propositional resource allocation.

2.1.1.4 Merging General Purpose Planning and Heuristic Planning

Although the efficiency of precondition planners increased successively since STRIPS, general domain independent planning procedures are still restricted to toy domains, due to the lack of suitable heuristics to guide search. Hierarchical plan component expansion is a computationally cheap alternative for the implementation of domain dependent planners.

Hierarchical plan component expansion, i.e., the successive refinement of plan components by subplans, is a heuristic planning technique. The planning process itself uses little, if any, search or backtracking to produce a plan.

This is in contrast to complete precondition achievement planners (“from first principle” planners), such as TWEAK, ABTWEAK, STRIPS, and ABSTRIPS which use exhaustive search and are guaranteed to find a plan, if one exist, even it may take very long.

O-Plan (Currie & Tate 1991) is a recent attempt in merging general purpose and heuristic planning approaches. It evolved from experience gained with earlier planning systems. O-Plan has been applied with several applications, such as oil platform construction project management, spacecraft mission sequencing, space platform construction, and software project management. However, most of the time it has been used as an off line plan construction tool rather than as an online task planner.

The system is rather complex and involves a lot of traditional planning by search, such as plan modification by extension, insertion, and reordering of plan components. Within this example, only the plan component representation and expansion are discussed as it is described as the feature that contributes heavily to the system’s high performance.

Particular emphasis has been put on limiting and guiding search during planning. The Task Formalism (TF) plan component representation of O-Plan consists of pattern-based descriptions, such as preconditions and effects, and additional information about plan component expansion and numerical constraints.

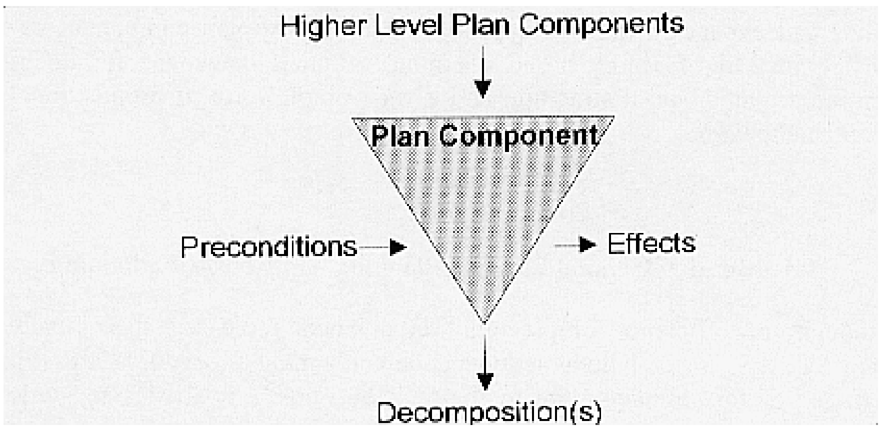


Figure 4. A Plan Component in O-Plan.

The TF-representation may represent plan components as either *block operators* or as *action-oriented operators*. O-Plan operators may be viewed as plan components extending plans both horizontally along the time axis and vertically down the decomposition abstraction hierarchy. Figure illustrates this view of plan components.

Block operators use a STRIPS-like plan component representation augmented by additional information to limit applicability and search during planning, while action-oriented operators are almost completely orthogonal to block operators. Whereas block operators enable search based planning, like the planning procedures discussed so far, action-oriented operators enable knowledge based planning through plan component expansion.

Figure presents an action-operator net in O-Plan (from Currie & Tate 1991, p. 61). In this figure, nodes not explicitly mentioned in the ordering statement can be executed at any time within the overall activity; they are parallel to those specified.

The condition types `supervised` and `unsupervised` are used to attach preconditions to subactivities which are either satisfied within the same overall activity (supervised conditions) or satisfied by activities outside the overall activity (unsupervised conditions).

The expressiveness of this kind of plan component description is very high and, in contrast to block operators, domain knowledge can be easily included.

```
schema build;
    expands {build house}
    nodes
      1 action {excavate, pour footers},
      2 action {pour concrete foundations},
      3 action {erect frame and roof},
      4 action {lay brickwork},
      5 action {finish roofing and flashing},
      6 action {fasten gutters and down spouts},
      7 action {finish grading},
      8 action {pour walks, landscape},
      9 action {install services},
      10 action {decorate};
    orderings
      1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6,
      6 → 7, 7 → 8, 2 → 9, 3 → 10
    conditions
      supervised {footers poured} at 2 from [1],
      supervised {foundations laid} at 3 from [2],
      supervised {frame and roof erected}
        at 4 from [3],
      supervised {brickwork done} at 5 from [4],
      supervised {roofing finished} at 6 from [5],
      supervised {gutters etc. fastened}
        at 7 from [6],
      unsupervised {storm drains laid} at 7,
      supervised {grading done} at 8 from [7];
    resources bricklayers =
      between 1 and 2 persons at 4
   (further components removed)
endschema;
```

Figure5. O-Plan Task Nets.

During planning an action like `build` is expanded into the sequence of actions given in its schema description. The corresponding subplan into which the action operator is expanded is given in Figure.

Another interesting knowledge based approach advocating the use of a very expressive operator description has been proposed by Dorn (1989, 1994). His research on dependable reactive event-oriented planning is a search-based hierarchical task decomposition planning approach using operators (scripts) which explicitly describe the properties, constraints, intervals, and causal relationships, in addition to the standard preconditions and results of operators.

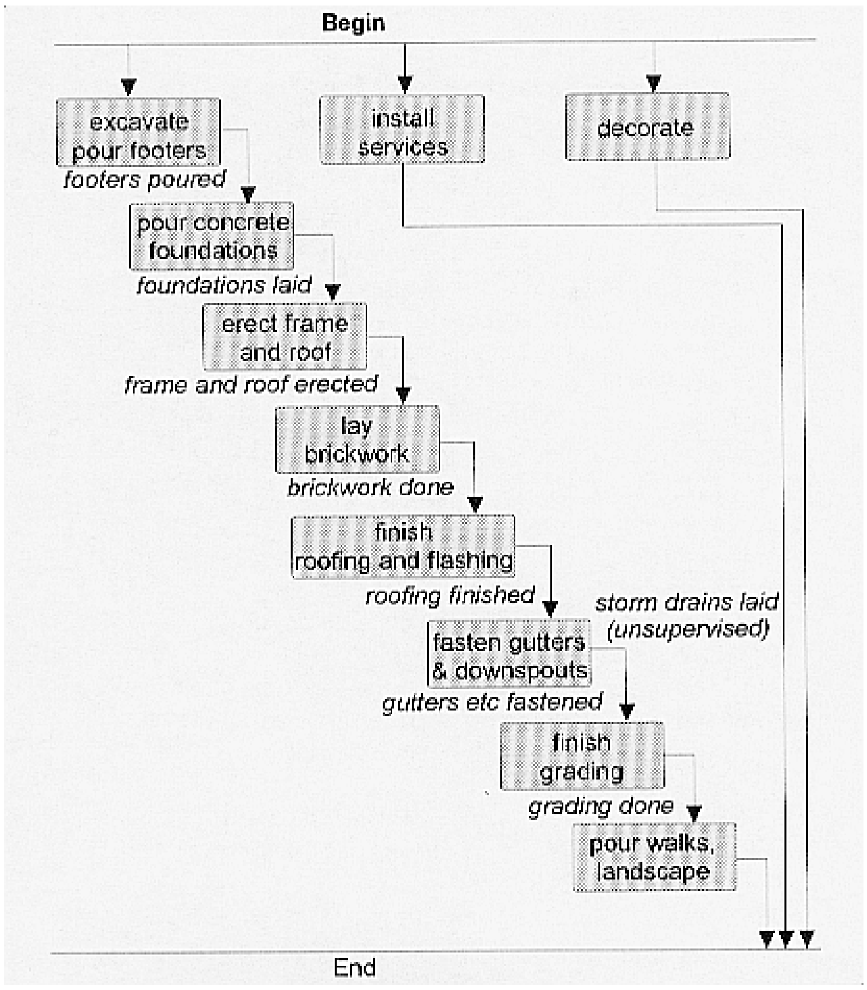


Figure 6. O-Plan Action-Operator.

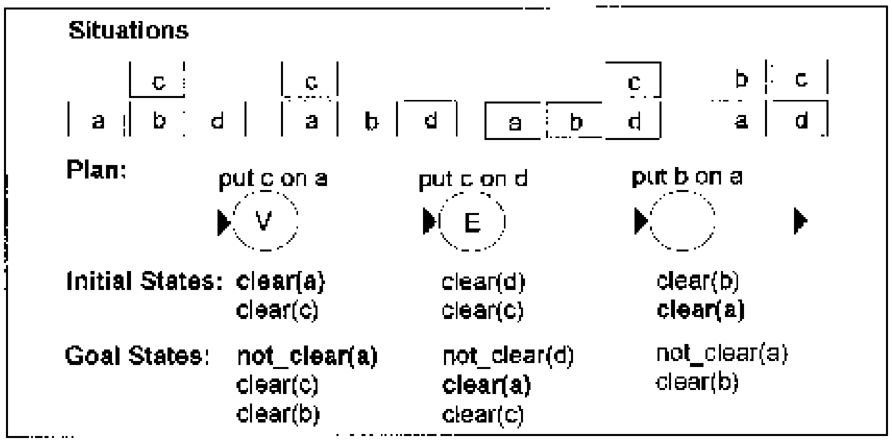


Figure 7. Violator (V) and Establisher (E) within a Plan.

2.1.1.5 Modal Truth Criterion - Plan Correctness

Whether or not a (partial) non linear plan is correct can be determined using the Modal Truth Criterion (MTC; Chapman 1987). This notation involves violators (also known as cloberrers) and establishers (also known as white knights), which violate or (re-) establish a certain proposition respectively.

Violators of a plan component *x* are plan components that are carried out before *x* and which undermine at least one prerequisite of *x*. Contrary, establishers of a plan component *x* are plan components that may be carried out before *x* and which establish a prerequisite of *x*.

Figure illustrates violators and establishers with respect to the proposition clear(a). The plan component marked with V is a violator of the last plan component with respect to the necessary initial state clear(a), which V violates. Luckily, for the sake of this example, this violated initial state (precondition) is re-established by the establisher E (which itself is also a violator of the final goal not_clear(a)), which clears block a by taking block c away from a and thus carrying the needed initial state clear(a) as a goal state in its plan component effect description.

The MTC reads as follows (Chapman 1987):

Definition 1. Modal Truth Criterion (MTC). A proposition p is necessarily true in a situation s iff two conditions hold: (1) there is a situation t equal or necessarily previous to s in which p is necessarily asserted; (2) and for every step V possibly before s and every proposition q possibly codesignating with p which V denies, there is a step E necessarily between V and s which asserts r, a proposition such that r and p codesignate whenever p and q codesignate. The criterion for possible truth is exactly analogous, with all the modalities switched (read "necessary" for "possible" and vice versa).

The MTC may be viewed as an undeterministic planning procedure, that is an arbitrary plan is taken and the MTC is used to check whether the plan is a correct plan. Consequently, the MTC (e.g. QA in NonLin and O-Plan) is a central algorithm within modern planners based on plan modification, such as NOAH, O-Plan, TWEAK, or ABTWEAK. The MTC necessarily needs to be checked to determine the correctness of a (possible partial and partially ordered) plan. Obviously, the MTC is exponential in the number of plan components and preconditions.

2.1.1.6 Complexity of Planning

The underlying representation of both the world and plan components' preconditions and effects directly influence the complexity of the underlying planning procedure.

The standard terms for describing complexity are P (polynomial time), NP (non deterministic polynomial time), PSPACE (polynomial space), EXPTIME (exponential time), and EXPSPACE (exponential space). P is the set of problems solvable in polynomial time, NP the set of problems of which a solution can be verified in polynomial time, PSPACE problems are solvable using only polynomial space for the computation, EXPTIME problems are solvable in exponential time, EXPSPACE demands exponential space. It is not known (but very strongly assumed) whether P is a proper subset of NP. NP is a subset of PSPACE which in turn is a proper subset of EXPTIME. EXPTIME is a subset of EXPSPACE.

A trade off between the expressiveness of the underlying framework, such as:

- the logic used to represent situations, preconditions and effects, e.g., propositional logic, first order logic with and without quantifiers, or Horn clause logic
- the plan representation, e.g. total order, partial order, conditional plans, loops, or recursion

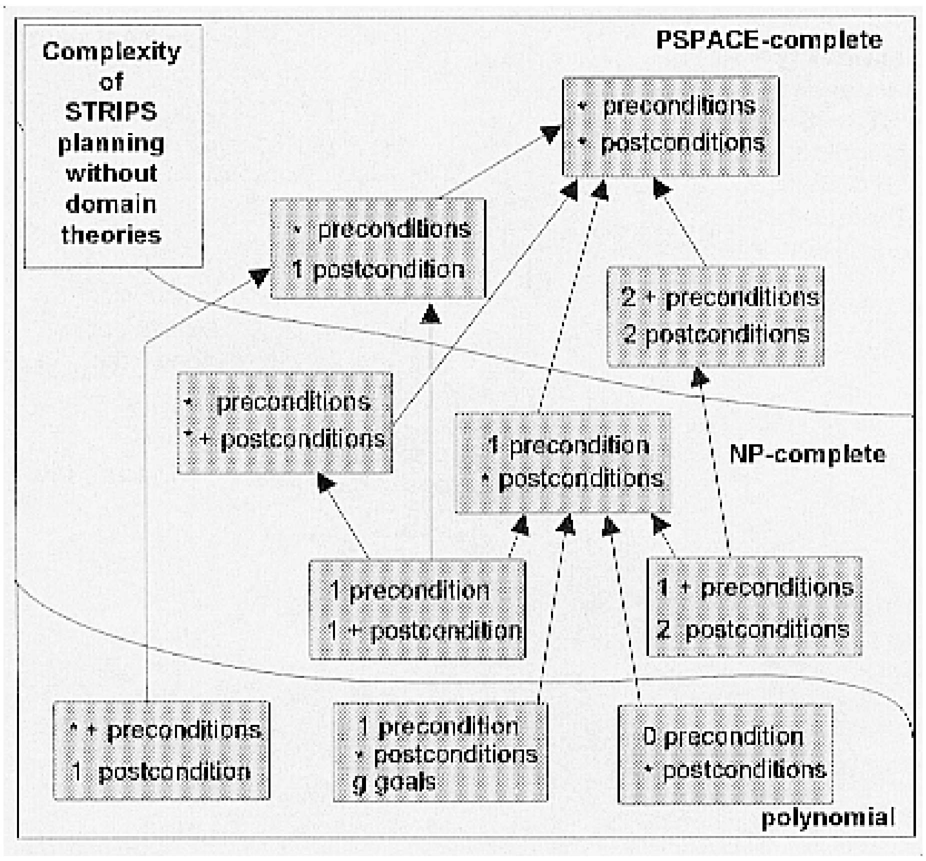


Figure 8. Complexity Results for Propositional STRIPS Planning without Domain Theories.

- additional domain theories to represent possible worlds (Ginsberg & Smith 1988),

and the complexity of certain problems must be considered. The complexity of precondition achievement planners under different restrictions is discussed below. The results of these complexity analyses for planning are disappointing, because for all interesting problems the run time behaviour of planning algorithms is exponential with the problem size.

Simple propositional planning is PSPACE complete (Bylander 1992) and planning with variables ranging over an infinite domain is undecidable (Chapman 1987).

Figure (adapted from Bylander 1992) shows the complexity of propositional STRIPS planning under different restrictions on the number of pre- and postconditions. In this picture “*preconditions” denotes no limit on the number of preconditions, “2 postconditions” restricts the number of postconditions to less than three, and “+preconditions” restricts preconditions to only non negated ones.

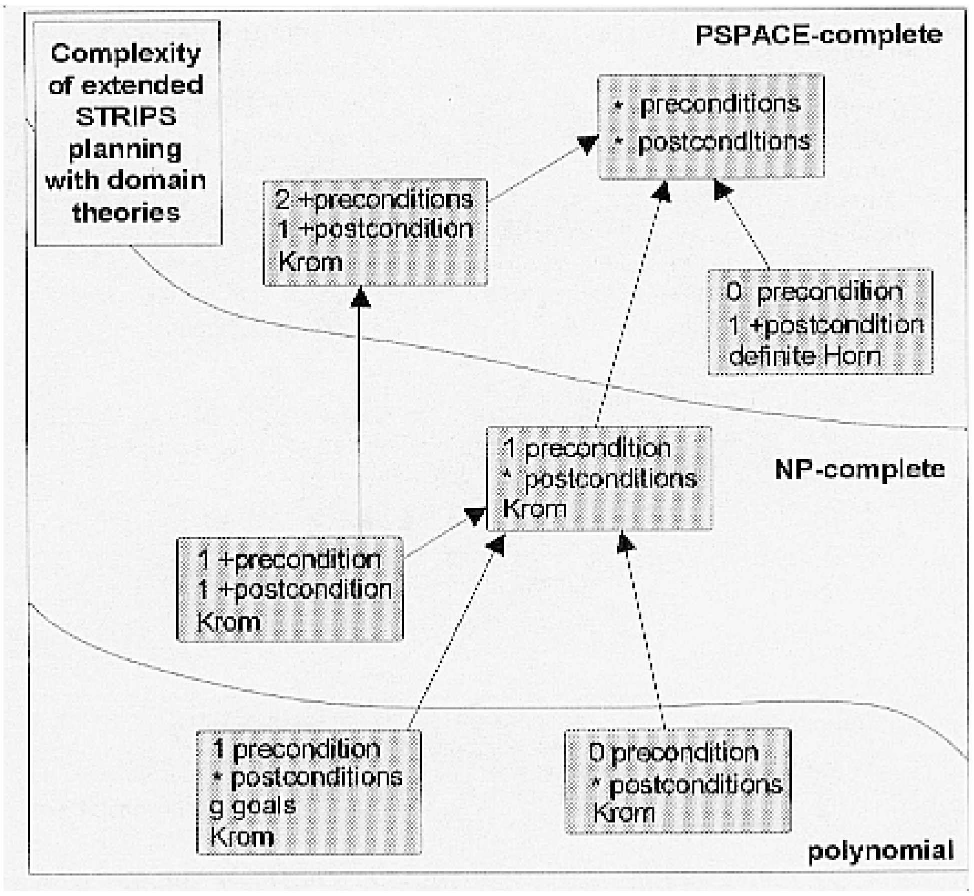


Figure 9. Complexity Results for Propositional STRIPS Planning with Domain Theories.

Since, propositional STRIPS planning is already PSPACE complete for the general case, only worse can be expected when extending propositional STRIPS planning to increase the expressiveness of the produced plans: Extending propositional STRIPS planning by domain theories, e.g., using first order logic without quantifiers, to represent knowledge within TWEAK makes the planning problem semi-decidable. TWEAK planning can be used to simulate a Turing machine, and the planner may not terminate if no plan exists (Chapman 1987). Domain theories form a set of propositional formulas such that all states within a plan must be consistent with the domain theory.

Domain theories can be used to represent global constraints and knowledge, such as “the ant-like robot may not lift all its legs at the same time”, “the torpedo must not be on collision course with one of our ships”, or “moving a block within a blocks world will move all block on top of it”. Figure summarises the complexity results on extended propositional STRIPS planning determined by (Bylander 1992). In this figure the same abbreviations as in Figure are used. Domain theories are either “Krom”, i.e. all formulas of the theory are a conjunction of two lit erals (Krom clauses), or “definite Horn”, i.e. all formulas of the domain theory are conjunctions of literals containing exactly one positive literal (Horn clauses).

PSPACE or NP-complete planning is not considered tractable for applications like robot control systems (or for any other application). However tractable planning, i.e. planning with polynomial complexity, may exhibit applicability for sequential control tasks within automatic control of industrial processes (Backström 1992). Naturally, the expressiveness of these restricted planning domains is limited and it is questionable if their expressiveness is strong enough for applications such as robot control.

Tractable planning has been investigated by (Backström 1992) and a class of planning domains has been found in which planning can be accomplished in polynomial time. Tractable planning in his approach is based on states described by an SAS^+ formalism. The SAS^+ formalism is similar to the STRIPS approach but differs in a few points:

- Multi-valued (a discrete domain of mutually exclusive defined values extended by the undefined value “u” and the contradictory value “k”) state variables are used instead of propositions to describe the preconditions and effects of plan components.

- A plan component description consisting of a list of pre-, post-, and prevail-conditions, instead of a precondition-, add-, and delete-list. The preconditions describe the conditions which need to hold before the plan component is executed, the prevail-conditions denote the conditions which need to hold before and during execution, and the postcondition specify the conditions which have changed as the result of execution.

- The following restrictions are given:

  a. The pre-, prevail-, and post-conditions are consistent.

  b. All state variables defined in the preconditions must also be defined in the postconditions, i.e., an action cannot change a state from a defined value to undefined.

  c. The prevail-conditions and the post-conditions must not assign different values to the same state variable. (A plan component cannot be requested to keep a value constant by one condition and to change it by another.)

  d. The pre- and postconditions must never have the same value for the same state variable. (If this is the case, those conditions should be modelled as prevail-conditions/transient state descriptions.)

  e. There is no plan component x for a plan component y ($x \neq y$) such that x and y have the same postconditions and that the preconditions and prevail-

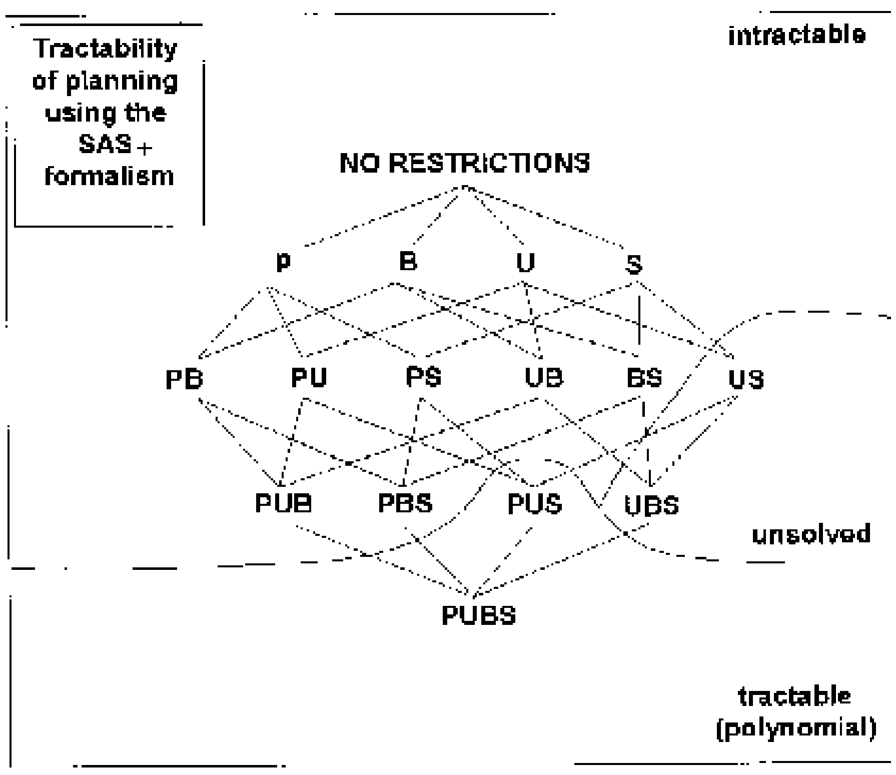


Figure 10. Tractability of Plan Search for the SAS− Formalism.

conditions of *x* subsume those of *y*. (Then the more restricted plan component would be redundant.)

- A number of optional restrictions are given as:

  (P) *post-unique*: No two distinct plan components can change the same state variable to the same value.

  (U) *unary*: Each plan component changes exactly one state variable.

  (B) *binary*: All state variable domains are binary.

  (S) *single-valued*: The prevail conditions of two distinct plan components must not require the same state variable to have different values.

Backström (1992) determined the complexity for plan search within the SAS$^+$ formalism for all combinations of the optional restrictions P, U, B, and S. His results are depicted in **Error! Reference source not found.** (adapted from Backström 1992). Only PUBS and PUS have been shown to have polynomial complexity for SAS$^+$, i.e. only when restrictions P, U, and S are met, plan search can be accomplished in polynomial time.

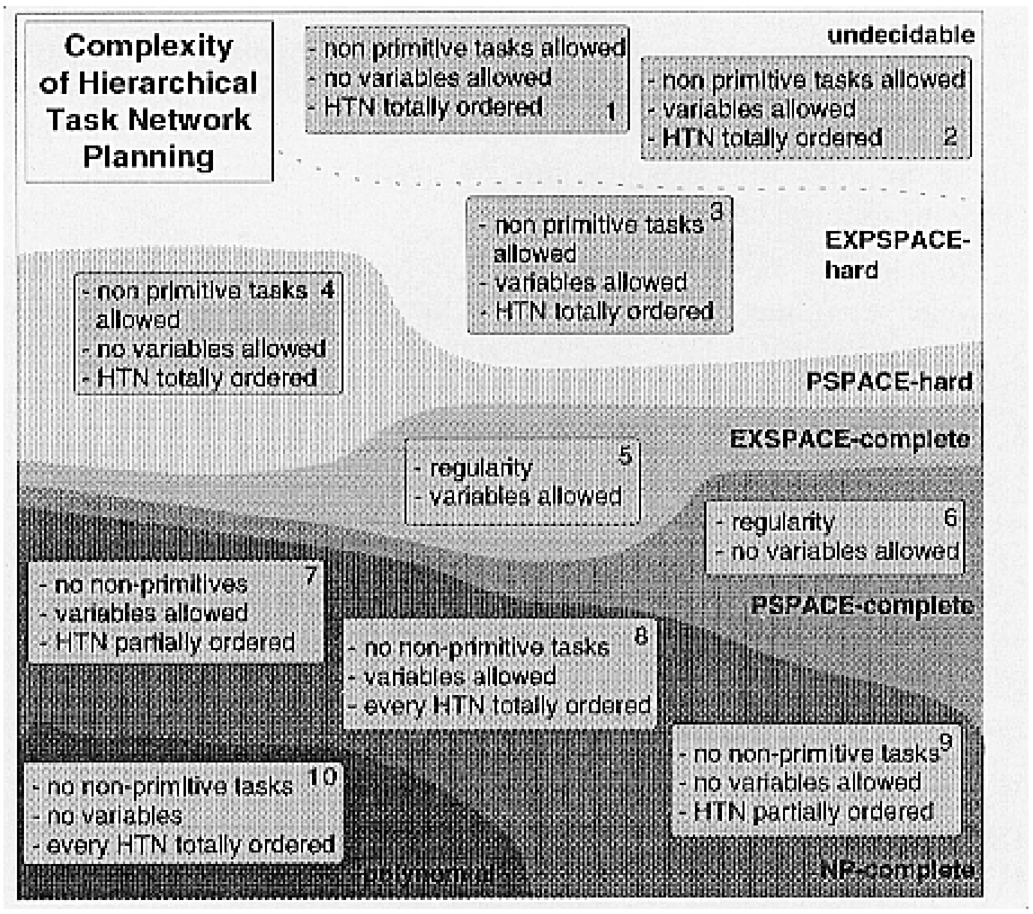

Complexity of Hierarchical Task Network Planning
undecidable
- non primitive tasks allowed - no variables allowed - HTN totally ordered 1
- non primitive tasks allowed - variables allowed - HTN totally ordered 2
- non primitive tasks allowed - variables allowed - HTN totally ordered 3
EXPSPACE-hard
- non primitive tasks allowed - no variables allowed - HTN totally ordered 4
PSPACE-hard
EXSPACE-complete
- regularity - variables allowed 5
- regularity - no variables allowed 6
PSPACE-complete
- no non-primitives - variables allowed - HTN partially ordered 7
- no non-primitive tasks - variables allowed - every HTN totally ordered 8
- no non-primitive tasks - no variables allowed - HTN partially ordered 9
- no non-primitive tasks - no variables - every HTN totally ordered 10
polynomial
NP-complete

Figure 11. The Complexity of Hierarchical Task-Network Planning.

Unfortunately, the expressiveness of tractable and complete planning methods is very limited. The most general tractable, planning method is SAS$^+$-PUS. Because of the low expressiveness, it may not be suitable as a general planning method but perhaps as a smart heuristic for arranging partial plans.

Most planners embedded in a real-world application are based on hierarchical task (network) decomposition. These were refered to above as “heuristic planners”. One reason for their use is their straight forward modelling of tasks and decompositions as long as domain specific (heuristic) knowledge is available; another the easy to implement planning mechanism.

Hierarchical Task Network (HTN) planning is based on successively decomposing (sub-) tasks into more specific task networks which are sets of tasks together with an ordering relation over these tasks. A typical representative of HTN planning is PEM (Planning, Execution, and Monitoring system) (Heikkilä & Röning 1992). The complexity of HTN planning has been investigated by (Erol et al. 1994b). A rigorous definition of the semantics of HTN planning can be found in (Erol et al. 1994a).

Hierarchical Planning: The complexity of hierarchical planning is (as always) related to the expressiveness of the plan representation used. Figure shows the complexity results from their research (adapted from Erol et al. 1994a). Situations 1 and 2, in this figure, are decidable if HTNs are acyclic and situation 5 is PSPACE if the planning domain is fixed in advance. The complexity of the planning problem (plan existence) was investigated under the following restrictions:

- **Restrictions on non-primitive tasks:** Non-primitive tasks are those which are further decomposed during the planning process (e.g. abstract plan components). They are either allowed to exist or not.

- **Restrictions on "regularity":** All task networks contain at most one non-primitive task which is ordered (in end position) with respect to all other tasks in the task network.

- **Restrictions on the use of variables:** Variables (e.g. planning variables, such as "x" in "GOTO x") may or may not be attached to tasks.

- **Restrictions on the Hierarchical Task-Network (HTN):** The HTNs to describe task decompositions may (zeroth order logic) or may not (propositional logic) be totally ordered.

One major result is that both the expressiveness and complexity of a general HTN-planner surpasses the expressiveness of STRIPS planner. On first sight this is quite astonishing since HTN-planners are the most frequently used planners for real-world applications, which should actually demand a low complexity planning solution.

However, for most applications HTN-planners have been chosen because of the possibility to express existing planning knowledge in a very compact way. This enables the HTN-planner to reason very quickly (in a straight forward model or schema based way) whereas the STRIPS planner or a non-linear planner inherently reason from first principles. Hence, the advantage of an HTN-planner compared to STRIPS-like planners are:

- Compact modelling of existing (model-based) planning knowledge using decomposition hierarchies. Hence, no need to plan for predefined plans or strategies which are well known beforehand.

- Fast (with respect to the amount of represented knowledge) instantiation based planning (with little backtracking).

- Easy maintenance and comprehensibility of planning knowledge.

From the theoretical point of view another major result of Erol's research is that HTN-planners can represent more planning domains than STRIPS-style planners. In fact STRIPS-style planning is a special case of HTN-style planning. The rela tion between them is analogous to the relation between context-free string languages and regular string languages.

2.1.1.7 Internal and External Plans

Up to now plans have been considered as plan component sequences to be executed by and under the supervision of an agent. These plans are denoted “internal plans”.

However, agents, the entities executing the plans, are usually not alone. Other agents, such as other robots, humans, or the “law of nature” agent may pursue their own intentions, i.e. their own internal plans. Plans of other agents are external to oneself and cannot be altered by oneself though they might be altered through “negotiation”. A conceptual “law of nature”, as a particular agent, is sometimes held responsible for all activities (surprises or unexpected events in general) within the world which are not in the scope of any modelled agent. Game theoretic approaches can be used to derive plans by playing against (or with) the “law of nature” agent

External plans may hold important information for planning, coordinating, and scheduling an agent's activities. Knowing the plans of other agents implies either communication about plans or plan recognition. This may be done by recognising conditions or execution of plan components that are almost certainly linked with the existence of an external plan, such as a flashing light indicating an automatic door is opening soon.

Plan recognition is a research area of its own and involves very specialised techniques. It has been applied in various domains ranging from military applications for determining the intentions of the enemy, to help systems for recognising the intentions of a user. An overview of different plan recognition systems and techniques is given in (Hecking 1993).

2.1.2 Low-level Planning and Control

Low-level planning and control techniques are characterised by the demand for fast response, the lack of highly assimilated information, and high frequency interaction with the environment. As pointed out above, planning and control span a continuum with no clear or obvious distinction between low-level planning and control discussed here and AI planning methods discussed above.

Whereas on the servo level linearity and continuity assumptions about the system often hold and traditional control techniques, such as PID controllers, can be employed, more complicated control tasks introduce non-continuities and non-linearities which make it difficult if not impossible to employ traditional control techniques. More elaborate system behaviours present a need of composing complex behaviours from simpler ones.

The simplest composition technique is concatenation or chaining, i.e. the sequential arrangement of behaviours with in plans, such as “pick up - move -place”. Blending and fusion are used for the creation of advanced behaviours by combination of simpler ones, such as “approach target” and “avoid obstacle”. Blending of behaviours combines low-level behaviours through the use of fuzzy logic, though other input-output mappings are possible, and results in the combination of output vectors of the same type, such as motor speed.

Plan merging (plan fusion) operates on plan structures and combines plans for multiple goals resulting in a combined, perhaps optimised, global plan.

In recent years many new ideas have emerged augmenting traditional control theory to bridge the gap between linear control theory and discrete control and planning, although they may focus on different aspects:

- **Intelligent Control Techniques** typically meaning Fuzzy Control (emphasis on representing rules of thumb control knowledge), Neuro Control (emphasis on learning) as emerging methods for controlling highly non-linear systems.

- **Linear and Non-linear Control Theory**, such as PID controllers or sliding mode controller, focus on modelling the dynamics and the mathematics for controlling a system.

The following techniques concentrating on behaviour fusion have been developed in recent years:

- **Subsumption Architecture**: composition of behaviours, layered organisation of competences using a concept of asynchronous model free augmented finite state machines. (Brooks 1989)

- **Potential Field Methods**: composition of behaviours by creating a gradient field of artificial force vectors overlaid on top of a spatial representation of the environment. (Khatib 1986)

- **Blended Behaviours**: composition of behaviours defined by fuzzy rule bases using concepts of applicability, desirability, and contexts as well as sensor and world model fusion using a spatial representation of the environment. (Saffioti et al. 1993a,b)

Composing complex behaviours by sequencing simpler ones results in plans with discernible components. Sequencing behaviours, which may involve plan generation using plan templates, is approached in different ways.

Due to the responsiveness required for low-level system components, sequencing is generally accomplished using (hand-written or off-line generated) plan templates stating in what order to execute lower level activities rather than by from “first principle reasoning”. In fact, hierarchical task network planning as described earlier is one of the different ways to approach sequencing.

Other appproaches of this type are Situated Automation Approaches (Kaelbling & Rosenschein 1989), Reactive Action Packages (Firby 1987), Universal Plans (Schoppers 1987), Teleo - Reactive Trees (Nilsson 1994), Procedural Reasoning (Georgeff & Lansky 1987), and Task Control Architecture (Simmons 1990, 1994). Due to their importance in Three Layer Architectures we discuss and compare these architecture in detail later.

2.2 MONITORING

Monitoring systems are meant to be watch dogs, taking care that system, environmental safety, and normal run of actions are ensured. They back up, assist, or replace human decision making in complex environments, such as nuclear power plants, chemical process plants, and other systems where the amount of information to be monitored is too extensive to be handled by human experts alone.

Monitoring is the task of comparing the actual system behaviour with the expected or planned system behaviour in order to detect deviations and errors. Monitoring systems increase aspects of systems' quality, such as enhanced autonomy and increased performance or efficiency. Models, which describe the system behaviour must be available in order to accomplish monitoring. Important properties of monitoring systems are:

- **Continuous Operation**: Monitoring dynamic systems, such as nuclear power plants, satellites, aircrafts, and surgical intensive-care-units, means continuously observing the system, timely detection of abnormal behaviour and system states which might cause failures in the near future and initial classification of these situations.

- **Real-time Constraints**: Monitoring systems must detect errors and deviations within certain time limits.

- **Embeddedness**: The hardware and software of the monitoring systems are intrinsically embedded in the system to be monitored, both physically (with sensors, communication, etc.) and conceptually resulting in a global surveillance system. For this reason monitoring systems cannot be viewed in isolation but rather in the context of plant control, diagnosis, and recovery.

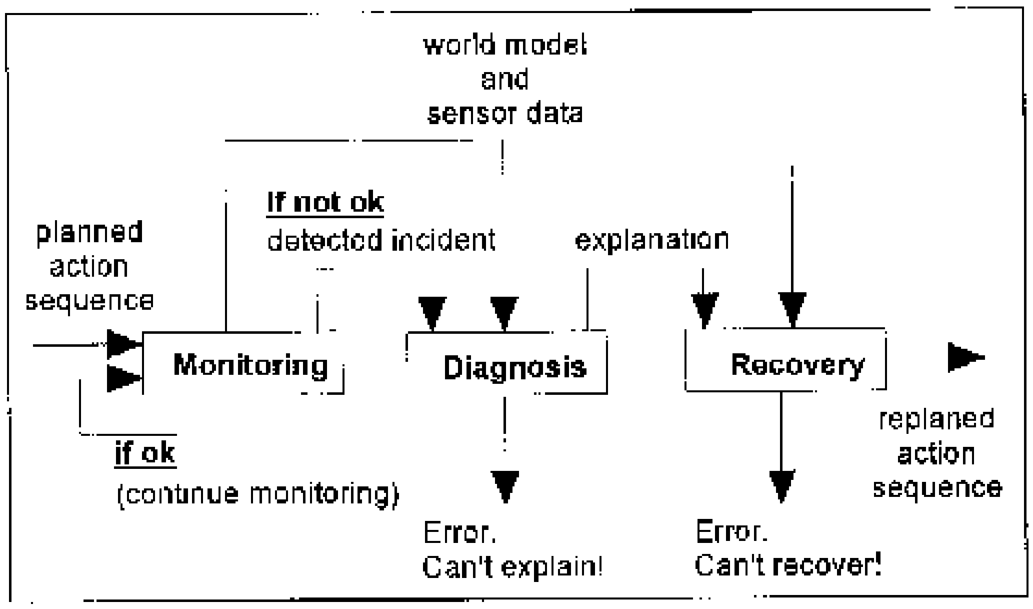


Figure 12. MDR Pipeline.

Monitoring systems are often conceptually embedded in the contexts of diagnosis and recovery as part of Monitoring-Diagnosis-Recovery (MDR) pipelines. Figure shows the stages in MDR pipelines which are incident detection, diagnosis, and recovery.

These stages may be fully automated or may include the human operator for problem identification and selection of corrective measures. MDR pipelines are related to control loops in the sense that both make decisions related to the system or its subsystems. Table 3 compares them with each other.

Table 3. MDR-pipelines and control loops.

MDR Pipelines	***Control Loops***
Responsible **for exception handling**: (non-nominal control flow) MDR pipelines are supposed to detect all out-of-specification behaviour, thus to increase a system's quality by adding robustness.	Determine the **system's normal behaviour**. Control loops incorporate dynamic features given by the system specification (nominal system control flow). Thus if following the specification, typical control loops ensure correctness.
Global Scope: MDR Pipelines are able to consider all information globally available to solve a problem, although it might be advisable to decompose the problem, i.e. to "divide and conquer" it.	**Local Scope:** Typical control loops use a constant set of continuously available information to control the dynamics of a (sub-) system. They operate locally.
Wide range of complexity or time frames: According to the nature of the detected problem, the amount of reasoning varies widely, implying varying response times.	**Constant complexity or time frames:** Due to the nature of control loops, their fixed set of inputs and outputs, complexity, and time requirements vary within limited bounds.
Open loops, may involve human interactions and are triggered by the detection of abnormal system behaviour.	**Control Loops are closed loops.** They continuously perform a mapping of input values (sensor readouts and history) to output values (actuator commands and feedback).

2.2.1 Monitoring Discrete Event Systems

Discrete event systems are dynamic systems with discrete, abrupt state changes, such as in digital electronic circuits, computer software, communication protocols, or intelligent robots at the symbolic task and operator description levels.

Various methods exist to describe such systems, including graphs, e.g. Petri nets, precedence networks, trees, and HTN plans, production rules, e.g. pattern triggered rule bases such as PRS (Georgeff & Lansky 1987) or a similar system proposed for manipulation planning within autonomous robots (Matsushita et al. 1993), finite automata, e.g. state transition diagrams, logic formalisms, e.g. situation calculus, time logic, and state charts, and symbolic plan and operator descriptions such as STRIPS. Common to these descriptions are states and state transitions, which in robot control architectures are induced by the execution of actions and deduced from sensed exceptions.

Monitoring discrete event systems implies the verification of pre- and post states (pre- and postconditions or effects) of state transitions (operators). However, the most common representation of intentions within robot control architectures are plans, which are partially or totally ordered sets of operators describing the sequence of operator execution. Plans commonly lack descriptions of how operator effects and operator preconditions are linked with each other, although this information exists during the planning process (e.g. as causal links) or in the mind of the system engineer who created the plan. Lacking this description will delay the detection of errors until the affected precondition is checked prior to the execution of the operator it is associated to.

One way to determine these dependencies is the Modal Truth Criterion (Chapman 1987). The Modal Truth Criterion as such does not cover possible effects and conflicts between concurrently executed plan components and needs to be extended to do so.

Both the use of the Modal Truth Criterion as well as the explicit representation of postcondition - precondition relationships within plan structures are advocated later in this thesis.

2.2.2 Monitoring Continuous Event Systems

Dynamic world descriptions are representation schemes which allow modeling dynamic changes in the environment, usually understood as derivatives of aspects, features, or values in time. With respect to this thesis, monitoring continuous event system is relevant to low-level physical actions in order to describe transient states, such as joint movements. Different techniques have been developed within the domain of qualitative physics to describe the behaviour continuous event systems, e.g., Qualitative Simulation (QSIM, Kuipers 1986).

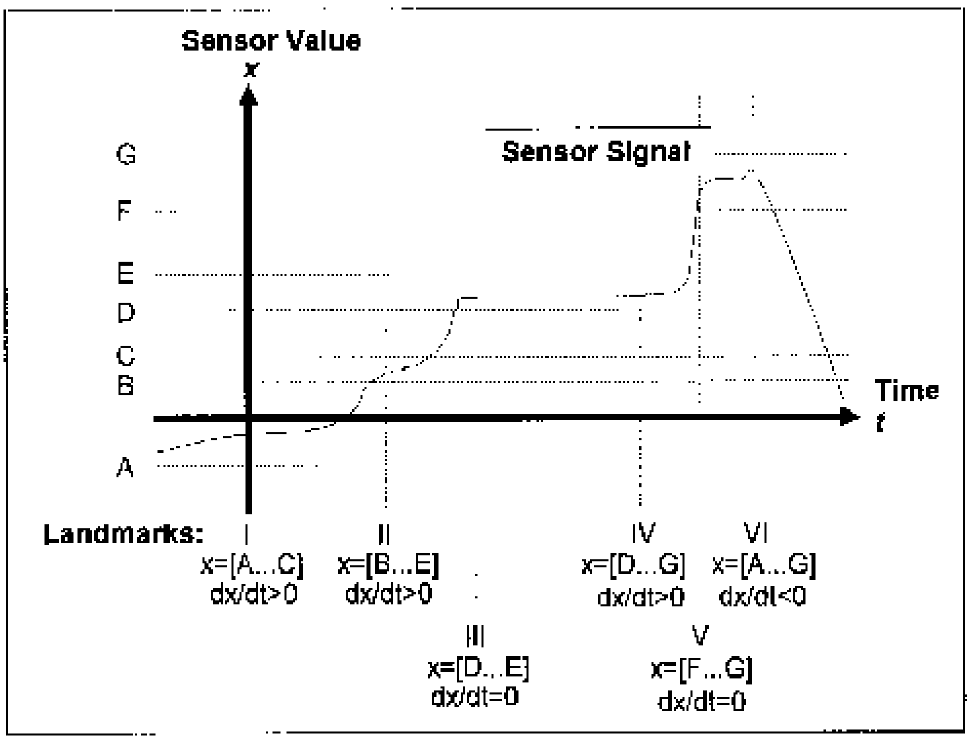


Figure 13. Sensor Signal Modelling using Qualitative Behaviour Descriptions.

The QSIM modelling approach (see Figure) is applicable for numerical sensor data, such as pressure, temperature, range, etc. Kuipers introduced a description language as well as reasoning techniques to match system behaviours with the system description in order to detect deviations. This description language consists of predicates which describe the behaviour of numerical values with time, based on derivatives. Lifting an arm from z = 3 to z = 6 could be modelled as z = [3...6] and z steadily increasing, i.e. dz/dt > 0.

He extended this concept with the notion of landmarks that define totally ordered intervals on the time axis and are used to delimit separate sensor value functions. A behavioural description of the total function is given as a sequence of partial descriptions. The resulting behaviour description of the function x given in Figure is: Behaviour(x) = ((x∈[A...C], dx/dt > 0), (x∈[B...E], dx/dt > 0), (x∈[D...E], dx/dt = 0), (x∈[D...G], dx/dt > 0), (x∈[F...G], dx/dt >= 0), (x∈[A...G], dx/dt < 0))

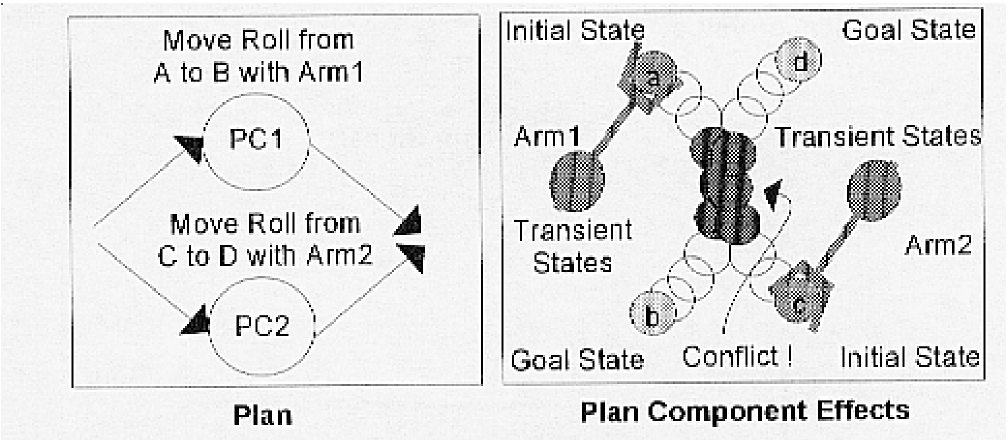


Figure 14. Concurrency requires checking for possible conflicts.

Figure 14. Concurrency requires checking for possible conflicts.

2.2.3 Concurrency and Partial States

Monitoring the execution and planning of concurrent activities involves partial world states. Partial states are subsets of the total world state describing distinct aspects of the world.

Concurrent actions must be checked for conflicts, i.e. the partial states describing the effects, preconditions, and transient states must not conflict as illustrated in Figure . To facilitate faster computation, conditions can be typed and assuming that conditions of different types cannot conflict, conflict detection can be parallelised by sets of World State Aspect Objects (Hasemann & Heikkilä 1993a,b).

Conflicts due to concurrency may not only occur between two plan components with conflicting transient states but also among multiple plan components. These conflicts are substantially harder to detect and involve representing constraints externally as part of a domain theory.

The following scenario illustrates a conflict involving multiple plan components. Consider an insect-like robot with six legs. Balance is possible as long as at least one leg on either side of the body and in total at least three legs are down. A situation where a conflict through interaction of plan components exists, e.g. when a plan includes three plan components each for lifting a seperate leg on the same side of the robot's body at the same time would cause the robot to tilt.

This conflict cannot be detected by consistency checking of the single effects and conditions of two involved parallel plan components, but involves an external formalism to reason about consistent sets of plan components. This external formalism could be a domain theory to represent global constraints and effects such

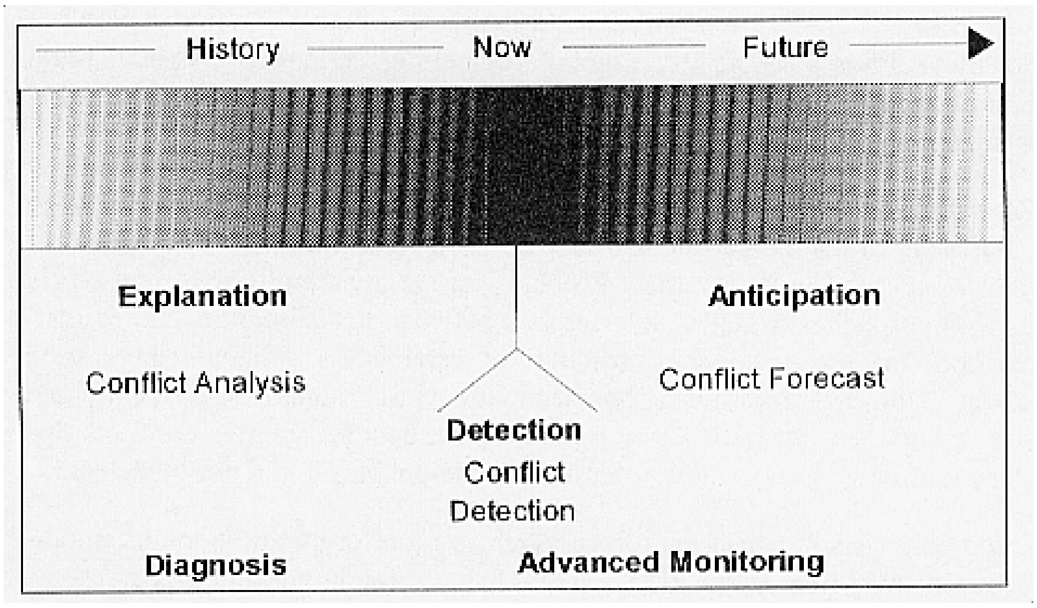


Figure 15. Monitoring.

as balance, gravity, or power consumption, e.g. a precondition generated during the planning process which requires two legs to be down on the side that is rising a leg and total of four legs to be down.

2.2.4 Monitoring within Robot Control Architectures

For robot control, monitoring is closely related to planning and execution. Monitoring is the task of continuously validating assumptions on the world during planning and execution. Planning, execution, and monitoring rely on data from the same or similar sources (e.g. sensors, world models, plans, etc.) and must respond in a timely fashion to data mutually received.

This property has been identified by Heikkilä and Röning (1992). In their approach, *planning*, *execution*, and *monitoring* are the generic activities within their robot control architecture (PEM architecture). Planning, execution, and monitoring are directly linked to the execution of single plan components. A PEM triplet combines the functionality of planning, execution, and monitoring for a particular action. It remains, however, questionable how plan validity monitoring, e.g. the surveilance of global dependencies, can be integrated within a hierarchical decomposition of planning-execution-monitoring triplets.

We distinguish between plan execution monitoring, denoting the validation of operator pre and postconditions and transient states, and plan validity monitoring, which denotes the monitoring of precondition - postcondition relationships. Plan validity monitoring can significantly increase performance of the overall system through shortening the plan-execution-replan cycle by early-as-possible initiation of replanning. Whereas plan execution monitoring can be considered as conflict detection, plan validity monitoring is concerned with future activities, i.e. conflict anticipation. See Figure .

For a long time, monitoring systems for intelligent robots were restricted to comparing measured sensor values with predicted ones, which indeed validates successful task execution. Probably due to the simplicity of real-world plans, approaches to detect interference between tasks being executed concurrently or the possible destruction of established subgoals have been scarce. Moreover, monitoring has been viewed functionally, i.e., locally and plan component centred. Hence, global dependencies such as conflicts due to concurrency or structural or temporal relationships could not be detected.

Little emphasis has been put on research in monitoring autonomous intelligent robots. Most of the research work on autonomous intelligent robots disregards continuous monitoring since other major problems related to intelligent autonomous robots seemed to be more urgent. That is probably why many proposed architectures for intelligent autonomous robots carry out monitoring by inserting perception requests into the planned sequence of action in order to verify pre and post-conditions. However, several monitoring systems for intelligent autonomous robots have recently been proposed, though most of them are restricted to plan execution monitoring.

Plan execution monitoring has been carried out in different ways. Doyle et al. (1986) proposed a system which automatically inserts perception requests into plans. Noreils and Chatila introduced Surveillance Monitors (Noreils & Chatila 1989, Noreils & Prajoux 1991) which are language constructs inside a plan that attach reflex actions to exceptional conditions. In contrast to perception requests, which are closely task related, Surveillance Monitors provide task-independent global condition checking.

The scope of monitoring consists of a sequence of actions (either atomic actions or Surveillance Monitors), during which the monitored conditions must be checked. The monitored condition is a logical conjunction of members, which specify sensors together with expected value ranges. If an error is detected, a sequence of Reflex Actions will be triggered. Reflex Actions are the system's direct response to the detection of an error. These Reflex Actions are either Surveillance Monitors (with the aim of monitoring a parameter more precisely) or simple commands (e.g. to slow down or to stop execution). Surveillance Monitors can also be declared as static, i.e. they remain enabled after detection of an error.

Surveillance Monitors are processed by a Surveillance Manager, which parses the monitoring conditions and passes its members (the permitted sensor value range) to a Sensor Surveillance Manager, which checks these members continuously. If a member is triggered, the Sensor Surveillance Manager which is associated with the responsible sensor, will inform the Surveillance Manager and the Surveillance Manager itself will evaluate the monitoring condition, and eventually alarm the Executive Module, which would execute the attached Reflex Action(s).

This approach has been improved by representing plans as finite state automata in which the outcomes (success or different exceptions) are directly connected to successive actions (Chatila et al. 1992). This monitoring system is able to monitor the execution of multiple concurrent actions. Emphasis is put on error recovery and a close connection between the monitoring system and the executing system. A finite state automaton approach is used, in which a set of possible results is provided to each monitor associated with an action. These results are “hard-wired” either with exception-handling routines, or in case of successful execution, with the next succeeding action.

Robot control architectures relying on AI planning techniques or representations rely on plan execution monitoring activities to validate correct plan component execution. A common approach is to validate pre- and postconditions of plan components at execution time, as done by GRIPE (Doyle et al. 1986). This system compiles a given plan, analyses it, and automatically inserts perception requests for monitoring. However, only pre- and postconditions are checked.

In another system (Miller 1989), an execution monitoring planner computes execution monitoring profiles based on a planned action sequence. The execution monitoring profiles are attached to the plan components and determine acceptable sensor readings which are monitored during execution. This planner roughly corresponds with the instantiation function described below.

However, little attention is paid to what happens during execution. This problem is addressed by Exception Handling Modules (EHM), which define tolerable sensor value ranges for physical sensors which are checked before, during, and after execution of a plan component (Meijer et al. 1990). EHM is a framework which integrates monitoring, diagnosis, error recovery and rescheduling. Its monitoring part uses sensor primitives which are attached to elementary operations (atomic actions). Sensor primitives represent tolerable value ranges for physical sensors. These sensor primitives are checked before, during, and after the execution of primitive operations.

Within architectures like the Task Control Architecture (Simmons 1990) or PEM (Heikkilä & Röning 1992), monitoring activities are defined explicitly and inserted into the plan structures. The TCA also monitored transient conditions.

Plan validity monitoring has so far not been in the focus of research, except for a monitoring system proposed by Reece and Tate (1994) and one proposed by the author (Hasemann 1992, 1994d), (Hasemann & Heikkilä 1993a,b), which laid the basis for the monitoring concepts described here.

Reece and Tate (1994) propose a monitoring system which generates monitoring requests from a non-linear plan representation and which also accomplishes plan validity monitoring has been developed independently. Based on and connected to a non-linear precondition achievement planner, the construction of causal structures for plan validity is guaranteed by the planners. Hence, the task of plan validity monitoring is significantly simplified.

Hasemann and Heikkilä (1993a,b) propose a monitoring system, JANUS, which accomplishes plan validity monitoring by incrementally evaluating the Modal Truth Criterion (Chapman 1987) for partially ordered plans over plan components which are represented as STRIPS-like operators (Fikes & Nilsson 1971) extended by a Transient States Description (Hasemann 1992). JANUS is described in more detail in Section 4.11.

2.3 INTENTION SWITCHING

Intention switching, that is abandoning or interrupting a currently active intention to engage another more urgent intention, has been investigated by a small number of researchers. Davis (1992) proposes a theoretical basis for interrupting and abandoning plans. However, it remains open how this approach can be extended towards hierarchical abstraction and multiple intentions.

Within robot control architectures, intention switching has been applied within in the Task Control Architecture (Simmons 1990) and the Procedural Reasoning System (Georgeff & Lansky 1987).

Intelligent robotic systems tend to be multithreaded similar to operating systems, by pursueing multiple goals with time-varying criticality. A robot may pursue different goals such as “recharge batteries”, “clean the living room”, or “serve pizza”. These behaviours have time-varying criticalities, i.e. “recharge batteries” is non-critical if the batteries are full. However, “serve pizza” may get a very high criticality if the operator has asked for it already twice.

Employing time-varying criticalities demands a mechanism for switching behaviours (Hasemann 1994a,b). As in operating systems, this mechanism is called behaviour switching. Goal selection, in fact, corresponds to the activation of a so far inactive behaviour. Intention switching is also important for carrying out global fault recovery.

2.4 FAULT RECOVERY

Especially in the real world, things naturally fail. Fault recovery is an indispensable task within robot control architectures. Research has so far concentrated on planning error detection and recovery strategies for motion planning (Donald 1987), (Latombe 1991). This thesis, however, emphasizes task planning aspects more than motion planning and we therefore concentrate on fault detection and recovery within task planning.

Reasons for failure and errors are manifold. Actuators are inaccurate, sometimes difficult to control, and sometimes break down. Similarly, sensors are imprecise and often subject to failure, ageing, drift, or simply breakdown. Complex interactions of physical system components may introduce sporadic errors. Sensor processing is limited by the available computational resources and the quality of the sensory input. Being situated in a world inhabited by other agents adds further possibilities of failures

One possible solution is to hardwire recovery actions to faults (Chatila et al. 1992). Another strategy for fault recovery is local recovery (Heikkilä & Röning 1992), which essentially means successively unrolling the planning process by falling back to more and more abstract levels of representation until replanning is successful. This strategy is local because it strictly follows the task decomposition tree set up during planning time.

This local strategy is inefficient and failure prone when faced with faults on a global level, e.g. a loss of hydraulic pressure renders all replanning of boom movements superfluous. According to the local fault recovery strategy the control system would unroll the planning process and fall back to higher levels of abstraction until it could finally change the motor setting to increase the hydraulic pressure. Local fault recovery in itself is simple, but unrolling the planning process through several levels of abstraction and then replanning is cumbersome and inapprobriate if the fault could be fixed by simple insertion of a recovery plan. This is called global fault recovery.

Global fault recovery is carried out by assessing the actual situation and trying to plan a sequence of operations (recovery behaviour) which changes the current (unexpected) state into the state which was initially planned for obeying constraints (expressed as virtual actions) which held at the time of recovery. This recovery planning can be assisted and guided by a central fault explanation system that tries to explain the fault based on fault patterns received from the virtual actions and initiates recovery behaviours in order to resolve the problem and return to the earlier planned sequence of actions.

2.5 DISCUSSION

Although the complexity results for all planning strategies are disappointing and at first glance discouraging, the situation is not really hopeless. Hierarchical task network planning, although at least NP-complete if variables are allowed, can efficiently produce complex expressive plans from a relatively small problem description. The reason for this lies in the action centered problem-specific operator description which uses different sorts of hierarchical plan templates (such as scripts and task nets) and instantiation techniques.

In contrast to these template-based planning techniques, traditional STRIPS-like planning techniques are prohibitively inefficient for use within a robot control architecture even when special techniques are employed to limit the search space or guide the search.

Monitoring has so far been restricted to validating preconditions, postconditions, and other explicitly specified conditions. For non-linear plans and STRIPS-like operator descriptions the modal truth criterion offers possibilities for the early detection of conditions which (may) interfere with later plan execution.

Intention switching introduces a straightforward way to independently plan for multiple goals and coordinate the resulting plans in a joint global plan. Finally, fault recovery is seen as a partly global and partly local activity which tries to repair faulty plans.

3 ROBOT CONTROL ARCHITECTURES

The previous chapter covered the theoretical background of plannning and monitoring. Within this chapter architectural aspects and applied technologies are addressed by a thorough review and comparison of proposed concepts and directions in the field of robot control architectures.

3.1 INTRODUCTION

Robot control architectures are sophisticated control systems with the purpose of enabling robots to do useful physical work in the real world. Figure depicts a very simplified view of robot control architectures. The robot control system continuously perceives the environment through sensors, updates an internal model of the world, deliberates, and passes actuator parameters to the hardware interface in order to pursue given goals.

During the last decade decisive progress in the field of robot control architectures has been made, though major problems still need to be addressed. The advent of the reactivity paradigm, with its most extreme implementation being the subsumption architecture (Brooks 1989), is probably one of the major developments in robot control architectures during the last 15 years.

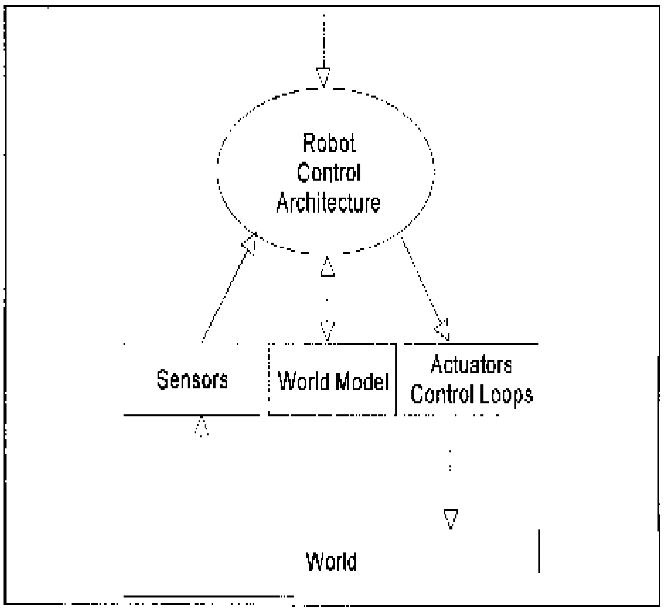


Figure 16. Simplified Robot Control Architecture.

Classical AI planning as the driving force in intelligent robotics up to the mid 1980's became augmented by reactive components and seems to have to lost importance because (a) most of the applications in intelligent robotics are very simple and can be sufficiently addressed by task nets or template plans rather than by planning from first principles, and (b) most pressing problems lie in the interaction of the robot with the world, the non-monotonicity of the world, and the apparent need for time bounded response at lower levels of representations.

Parallel to the development of robot control architectures, the physical components of intelligent robots became cheaper with much better performance. This applies particularly for sensors, actuators, drives, light-weight structures, microprocessors, and other electronic components. Other contributing factors are advances in packaging techniques and in the integration of electronic and mechanical components, i.e. mechatronics.

These two developments make intelligent robotics more and more attractive for application areas in addition to the prior planetary, deep sea exploration, and various military application areas. These new fields of application are very generally subsumed by the term service robotics. Service robots are (usually mobile and portable) intelligent robotic systems distinguished by the nature of their tasks, i.e. services, and by their mobility or portability from their relatives which operate in confined spaces within manufacturing processes. Service robots are defined by their functionality and, not surprisingly, are emerging from industry-led research projects, such as the European Community research projects MARTHA, NEUROBOT, PANORAMA, and others (EC 1994).

Examples illustrating the broad spectrum of existing and envisaged applications of intelligent robots are mobile platforms for in house courier tasks (e.g. HelpMate by TRC); brick-laying robots; cleaning robots for offices, public places, and private houses; car disassembly robots, kitchen robots for catering companies; and many others. For an overview of service robot applications see (JIRA 1994), (Aaltonen 1993), (SR 1993), (FhG-IPA 1994a,b).

Other sophisticated applications being researched arise in the traditional research fields of intelligent robots such as planetary, volcanic, and deep sea exploration as well as robotic applications in space or on the moon.

In the following section we investigate the dimensions and attributes of applications which can be used to determine the complexity of an application in a qualitative manner. Moreover, we review the development of robot control architectures below, discuss the directions, point out some of the major properties of the different directions, and then look at applied technology.

3.2 DIMENSIONS AND ATTRIBUTES OF APPLICATIONS

Performance of a robot control architecture depends on various parameters, such as the complexity of the task or the environment. Metrics or benchmarks are very difficult to establish for two reasons (Drummond & Kaelbling 1990).

First, because robot control architectures operate in dynamic, only partially structured, environments which even for the sake of simulations can only be partially modelled. The dynamics of reality under which a robot control architecture should be tested in order to achieve a good benchmark are not repeatable.

Second, the application bandwidth for robot control architectures is too broad to be represented by a single benchmark and depends to a large extent on implementational issues and matters of available tools and manuals.

Nonetheless, we try to list attributes which when existent make an application significantly more complicated from the point of view of the architecture. We call these attributes dimensions since they collectively span the design space for robot control architectures. Within this design space certain trade offs, illustrated below, can be made in order to achieve feasible solutions.

We identify the following dimensions by which the complexity of applications for intelligent robots can be estimated:

- The **level of autonomy** can roughly be described as the percentage of unattended operation without user intervention. The level of autonomy corresponds to Sheridan's spectrum of control modes (Sheridan 1992). A low-level of autonomy denotes a (slave like) system almost directly and continuously controlled by its master (usually a human operator) whereas a moderately autonomous system is under supervisory controlled: it operates largely unattended and only occasionally needs user intervention.

  The extreme stance is total autonomy with hardly any or no user intervention. Examples for this are space probes, which occasionally need ground control intervention, and Brook’s animats (Flynn & Brooks 1989), which fall most closely into this category. Upcoming applications of animats are autonomous planetary exploration, surfzone mine hunting robots, pipe inspection, and concrete trowelling (IS Robotics 1995). A high-level of autonomy implies sophisticated, not necessarily computational expensive, error recovery strategies.

- **Environment complexity:** (including the perception thereof) Environmental complexity is low when the environment is largely static (no changes in the environment other than those planned for), well structured (easily modelled), and easily and reliably perceived (no mismatch between the world and its model), whereas highly complex environments are dynamic, less structured, and difficult to perceive.

- **Task complexity:** the amount of planning, error recovery, choices, and decisions to be made. Low complexity tasks are very simple with only one or a few different ways (functional redundancies) to tackle a problem. Spotwelding chassis in car manufacturing is an example of a low complexity task, since the robot -- automaton like -- performs the same pre-programmed action sequences time and time again. Not surprisingly, its environment is at the same time very simple and predictible, thus of low complexity as well.

Environmental complexity and task complexity define the “sophistication” of a robot’s job.Very often, they are correlated because of uncertainties of sensors and actuators. Misperception, execution errors, and changes in the environment affect task execution and call for attention. Thus they increase the frequency of exceptional situation which are addressed by recovery activities which in turn further add to the task complexity.

Other view of this complexity space include (Drummond & Kaelbling 1990):

- Provision for **Specialised Reasoning**, such as geometric reasoning (e.g. path, and motion planning), temporal reasoning, scheduling, etc.

- **Learning Capability**: Should the robot be able to learn from experience or is the provided information sufficient to fulfil the assignment?

- **Multiple Agency**: Should the robot be able to interact with other robots and humans and if so to what extent?

- **Type and Amount of Domain Knowledge**: How much and what kind of domain specific knowledge (CAD-data, maps, models, task decomposition schemes, etc.) must be added to the system to be operational?

- **Informability**: Should the robot be able to respond to new facts and goals presented during the course of execution, i.e. is there a need to be able to revise or optimise the current plan or to interrupt the current plan to pursue a more important job? Is there a need for monitoring plan execution and plan validity? Are explicit sensing strategies necessary to compensate for information decay (Schoppers 1987) in the world model? Should the robot take advantage of opportunities arising during execution? Should the robot be able to interrupt a current task in favor of a more urgent or critical one?

- **Resource and Deadline Management**: Should the robot be able to optimise resource usage? Does the robot need to obey strict deadlines?

Important to notice is the strong correlation of two of these task attributes. Informability and multiple agency are attributes which more or less imply each other. As illustrated by Firby (1995) tasks such as vacuum cleaning can be made arbitrarily complicated by introducing or relaxing constraints on the environment or the task, such as possible interruption by a human, moving obstacles, arbitrarily shaped obstacles, etc.

The broad spectrum of driverless transport systems demonstrates how a simple task such as transporting material can cover the whole spectrum from very simple, e.g. contemporary AGVs guided by cables in the floor, with right-of-way along their tracks and controlled by programmable logic controller, to very complicated, e.g. AGVs in less structured and dynamic environments with navigation and obstacle avoidance accomplished using ultrasonic transducers.

That said, the level of autonomy plus task and environmental complexity constitute design space for intelligent robots. Compromises must be made by structuring the environment or reducing the robot's level of autonomy in order to keep the overall complexity at a reasonable level. Possible trade-offs are:

- **Reducing the level of autonomy**, e.g. by concentrating on routine time consuming tasks and delegating more difficult tasks or decisions to a human operator.

- **Reducing environment complexity**, e.g. developing intelligent robots for cleaning public places, aeroplanes, trains, toilets, or offices may imply changes to the environment to make it more robot friendly by introducing landmarks for navigation, interfaces for lifts, and changes due to the robots physical capabilities, e.g. specially designed seats in aeroplanes, staircases at railway stations, etc. (FhG-IPA 1994a).

3.3 DESIGN CONSTRAINTS AND GOALS

Designing control systems for a robot just to survive in an environment not designed for them is difficult for several reasons (Gat 1991):

- **Timeliness of decision making and acting:** Time available to decide what to do is limited. Due to time constraints only limited amounts of world knowledge can be assimilated and used for controlling the robot. Under certain conditions decisions must be made quickly on the basis of slightly processed “raw” but “recent” information. The utility of information increases through assimilation but decreases with the time needed for assimilation. An optimal trade-off between the two must be found.

- **Unpredictability of the world:** The world is to a large extent unpredictable. It is dynamic in that it changes without the robot's intervention. It is adverse at worst and indifferent at best. Events occur suddenly and demand rapid attention and response by the robot.

- **Imperfect sensors and actuators:** Sensor and actuators are prone to failure and breakdown. Sensors have only a limited range and resolution and may emphasise environmental “noise”. The perceived image of the world is at best a good approximation of a portion of reality. Actuators are limited in their capabilities and have only limited accuracy.

Further design constraints enabling the robot to do useful work in an efficient manner are:

- **Robust behaviour** in spite of sensor noise and actuator errors

- **Flexible, goal directed behaviour**, i.e. purposeful “intelligent” task solving strategies.

- **Usage of (domain) knowledge** at various levels of abstraction

- **Reactivity:** Rapid response to unexpected events

- **Reasoning about goals, plans, and the environment** (such as planning, scheduling, monitoring, task criticality, task switching, and symbol anchoring)

3.4 DIRECTIONS IN ROBOT CONTROL ARCHITECTURES

Throughout the years of building robot control systems a number of paradigms, good practices, design directions and reference architectures emerged. Some of the more commonly known paradigms are listed here for convenience.

Sense-Model-Plan-Act: The sense-model-plan-act paradigm (Figure) describes any control system in a nutshell. Although the traditional sense-model-plan-act paradigm of traditional robot control architectures has been much criticised, it is an essential part, in one form or another, of all mainstream architectures, including NASREM, Subsumption, RAP, Teleo-Reactive Programs, and PRS, though the differences in realisation can be huge, e.g. between Subsumption and NASREM.

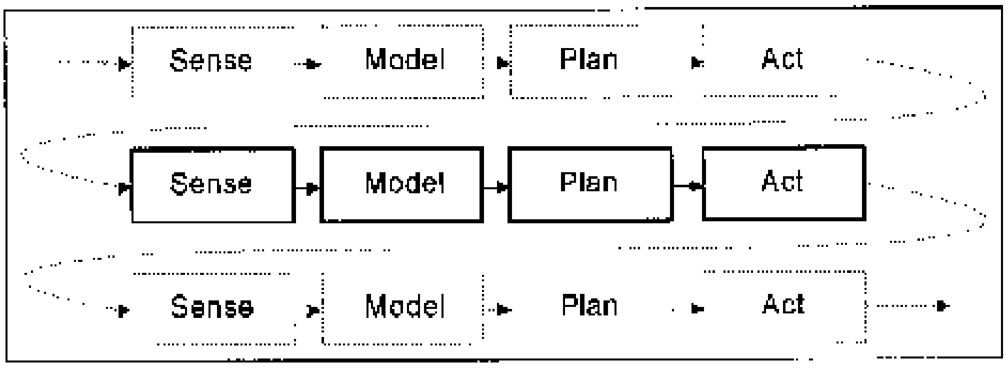


Figure 17. Sense - Model - Plan - Act Architecture.

The extent, however, to which the activities of sensing, modeling, planning, and acting are instantiated in particular systems varies greatly. Specifically, the amount of deliberation, i.e. modelling and planning, can be strongly emphasised as in expert-system control or totally neglected as in mechanical control systems as simple as thermostats. Nevertheless, acting should naturally depends on what is sensed and to varying extents on its assimilation into models, sensor data, and derived intention structures. Actions can be as simple as single action commands or as complicated as conditional non-linear plans.

The sense-model-plan-act paradigm assumes a quasi-static or at least predictable world assumption between sensing and acting. As this is a rather invalid assumption for most real-world domains, approaches have been undertaken to improve reactivity by breaking up the original single sense-model-plan-act execution pipeline into a number of parallel and interleaved ones. This has led to decisive advances by exploiting both hierarchical structures and parallel concurrent structures.

Hierarchical or vertical decomposition, a common paradigm in many application areas, splits the robot control system vertically into levels of hierarchies (as in NASREM, PEM, Subsumption, ...) whereas horizontal or lateral decomposition introduces concurrent control entities (called behaviours, reactive action packages, or teleo-reactive programs) to split the robot control architectures horizontally into concurrent operations.

Hierarchical Decomposition is based on the assumption that the dynamics of the world decrease with the level of abstraction. The hierarchical decomposition paradigm implies the subsequent decomposition of tasks into subtasks of the next lower level. The control flow (initiation, termination of subtasks) is up-down and the data-flow (execution results and sensor readings) is bottom-up, via assimilation of sensor data, sensor fusion, and world modelling. Levels of low abstraction such as servo control are recognised by high immediacy and depend on little or slightly assimilated information. Higher levels, in contrast, may depend on highly

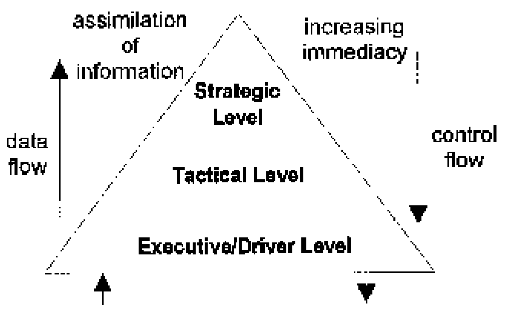


Figure 18. Hierarchical System Decomposition.

symbolic assimilated data. Reasoning on higher levels results in plans and resource allocations with long durations (Albus et al. 1989). (Figure)

Hierarchical decomposition is particularly attractive due to the improved combinatorics of information extraction, both the assimilation and caching of widely used pieces of information. Drawbacks of purely hierarchical decomposition lie in the separation of information acquisition and usage, causing loss of efficiency. There are also engineering related drawbacks such as early definition of interfaces between modules and layers potentially impeding expandability in later stages.

Lateral Decomposition denotes cooperative decentralised problem solving on each level of abstraction (if any). Examples for lateral decomposition methods include blackboard systems and most reactive approaches described below. They have in common the existence of methods to exchange information, though different in their motivation, approaches, and realisations. (Figure)

Reactive Systems Metaphor (reactivity): Reactivity commonly denotes either

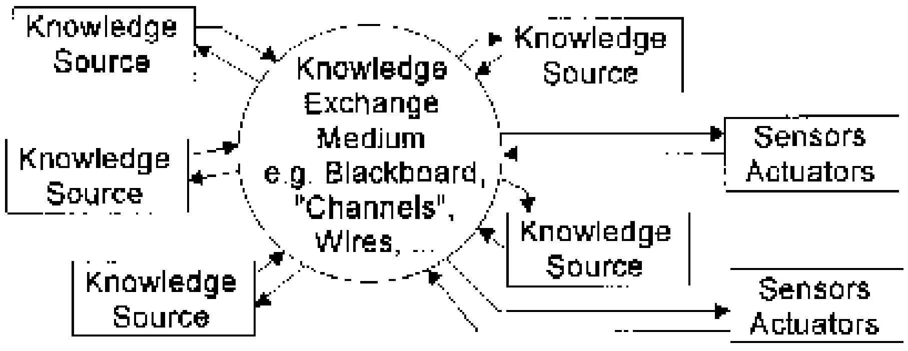


Figure 19. Lateral (blackboard) Decomposition.

(a) the *appropriate speed of reaction*, or (b) *minimal usage of internal state information* (Gat et al. 1994). Systems which share both properties are said to be reflexive.

- "Appropriate speed of reaction" is, however, situation and task dependent and in itself just another design parameter.

- "Minimal (or no) use of state information" implies that behaviours tend to be little or not at all goal directed. Neglectance of state information, in particular world models, makes it difficult to enable look ahead planning and impairs goal-directed behaviour. Very simple computational units can be used which ensure rapid computation and response to external triggers enabling robots to survive (at least in the short term) in dynamic environments.

Different approaches have been developed to create "reactivity" on different level of abstraction. Among these are the *subsumption* (and similar) *architectures* (Brooks 1989), (Gat et al. 1994) and *blended behaviours* (Saffioti et al. 1993a) for the lowest (non-symbolic) level of abstraction and *reactive planning* (Firby 1989), (Georgeff & Lansky 1987) and *compiled plans* (Nilsson 1994), (Schoppers 1987), (Kaelbling & Rosenschein 1989) approaches for the next higher level of abstraction.

Subsumption Metaphor: Subsumption architectures are collections of simple behaviours which act as building blocks for complex behaviours. Behaviours are based on augmented finite state machines, organised in layers of competences, and interconnected by "hard wired" connections to inhibit or enforce each other. The creation and maintenance of an accurate world model is time-consuming and thus avoided. In its purest form, subsumption architecture does not allow internal states, thus behaviours merely appear to be transfer functions from sensory inputs to control outputs. Subsumption mechanisms are widely used to implement low-level reactive behaviours such as walking and obstacle avoidance (Brooks 1989) in hybrid architectures.

Drawbacks of subsumption like architectures are in particular the arbitration required among multiple, potentially inconsistent, behaviours. Moreover, subsumption architectures often require major redesigns to change behaviours. Their performance is weak if relevant information is not locally available through sensors, and thus should be limited to low-level behaviours.

Aside from the original and purest form of subsumption logic several variants have been proposed. These differ in details such as the use of internal states and the type of communicated information. A suite of languages has been developed to facilitate the design of subsumption (and similar) architectures, e.g. REX (Kaelbling & Rosenschein 1989), ALFA (Gat et al. 1994), and Behaviour Language (Brooks 1989).

Several systems emerged adopting a *circuit metaphor* for subsumption-like controller structures. Computational units are finite state machines which may incorporate internal states. This is discouraged by Gat et al. (1993). Communication is in contrast to the pure subsumption architecture not limited to binary signals and states but arbitrary information can be exchanged through channels between adjacent state machines.

Reactive planning approaches are recognised by the use of plan templates for task decomposition and heuristics to select among different possible instantiations. Plan templates can be very complex, including parallelism and choice. For this reason they are often called procedural knowledge. Two representatives of this class are the procedural reasoning system (PRS) (Georgeff & Lansky 1987) and reactive action packages (RAPs) (Firby 1989) (described below).

Compiled plans approaches consider all possible situations (or at least a large subset of them) during the design phase and map appropriate plans to them instead of using explicit run time planning to form a plan to reach a goal state from the current situation. Examples of compiled plans approaches are teleo-reactive trees (Nilsson 1994) and universal plans (Schoppers 1987). A special type of compiled plans approach is based on the *situated automaton* theory developed by Kaelbling and Rosenschein (1989).

Situated automatons are used in a formal methodology for constructing robot control systems. In its general form the methodology is based on finite state machines which receive sensor input via an update function and have an action function for physical interaction with the environment. The execution cycle consists of reading the sensor input, updating the internal state, and the (time-bounded!) calculation of the output vector. Although internal states exist these are to kept to the absolute minimum necessary for carrying out the tasks. Since situated automaton theory is a formal methodology it is not restricted to any kind of architecture. However, various support tools have been developed to create reactive systems based on situated automaton theory, e.g. GAPPS (Goals as Parallel Program Specifications), RULER, and REX. In its earliest form GAPPS took top-level goals and goal reduction rules as inputs and creates "circuit" with situation specific responses. Goal regression was added later. One disadvantage is that GAPPS lacks the ability to express procedural information, such as action templates.

Universal Plans (Schoppers 1987) are decision trees which map the current world state into the next action to take. They are created off-line by a "reverse-planning" procedure which takes a goal state condition (note: no initial state condition) and a set of operators as inputs and generates a Universal Plan by back-chaining from the goal condition using the effect descriptions of the operators. Although Univer sal Plans prove to be very fast in selecting the optimal action to select, their application to real-world problems is problematic to due the exponential growth of the decision trees as the numbers of actions and predicates increases. Nevertheless, the idea may well survive as almost-universal plans covering almost all possible world states leaving the remaining situations covered by "deliberative methods" (Schoppers 1989).

Blended behaviours (Saffioti 1993a) constitute a new direction and are based on fuzzy rule sets and fuzzy logic composition rules in order to blend simple behaviours to form complex ones. Context-dependent blending of behaviours is accomplished using (fuzzy) desirability functions and context rules. This composition of behaviours is superior to composing behaviours using "potential field" and artificial force fields (Latombe 1991) since not all constraints can be easily expressed as artificial force fields. Moreover, this approach facilitates easier interaction with the next higher level in three layer architectures because of the direct correspondence of low-level composition rules of fuzzy behaviours (= rule sets) with the composition mechanism used in symbolic layers, i.e. conjunction, disjunction, and sequencing.

Belief Desire Intention (BDI) Metaphor (Sundermeyer 1993) uses the explicit representation of beliefs, desires, and goals within the agent's or robot's control architectures. Beliefs, roughly corresponding to a world model, are usually represented by a propositional database which can be questioned by other subsystems. Desires are the current goals of the robot; and intentions denote the representation of what to do next. Intentions are usually represented as plans. BDI architectures also imply that the robot is committed to its intentions. Because of the explicit representation of beliefs, desires, and intentions, BDI architectures are suitable for multi-robot applications in which it is required to have a means-ends analysis based cooperation mechanism such as contract nets or game theory. Example BDI architectures are the Procedural Reasoning System (Georgeff & Lansky 1987) and the Reactive Action Packages (Firby 1989).

Homeostatic Control was first introduced in AuRA (Arkin 1989) and is motivated by the endocrine system known from anatomy. The function of homeostatic control is to monitor safety critical system aspects such as battery charge level, vehicle balance, etc. and to eventually trigger corrective activities or change task criticalities. Although some safety-critical aspects such as obstacle avoidance were earlier covered by reactive subsystems, others which demand more analysis for detection (such as prospective fuel consumption with respect to a current plan) or reaction (setting up a plan to the next fuel depot) had not been addressed. Homeostatic control can be viewed as a special instance of task switching.

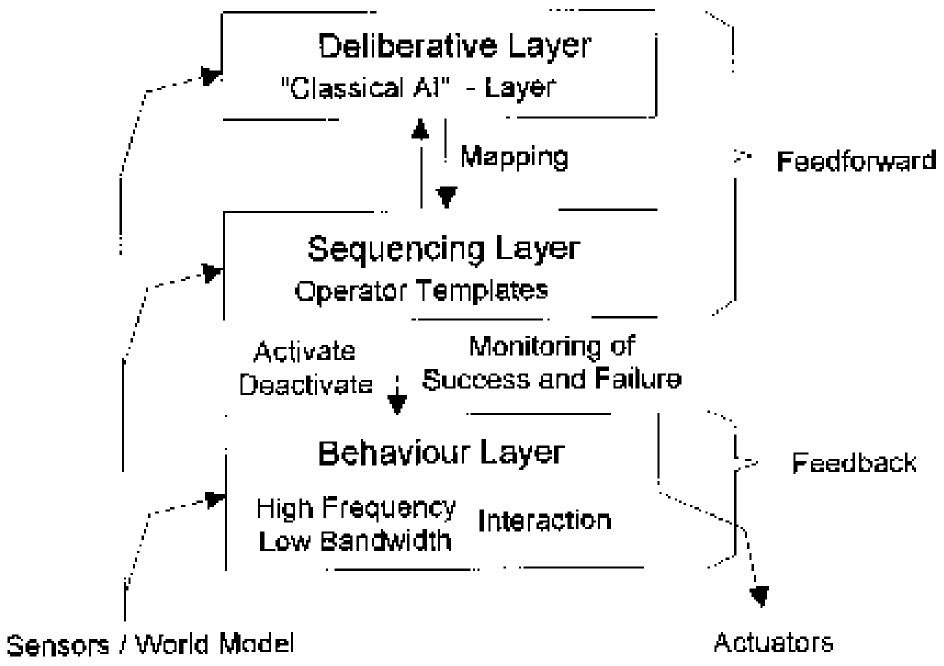


Figure 20. Three Layer Architecture.

Hybrid System Metaphor: Several architectures, called hybrid architectures, have been designed to overcome problems inherent in traditional hierarchical architectures and reactive architectures by blending deliberative planning, i.e. symbolic representation and reasoning, and reactive mechanisms in order to overcome their separate disadvantages.

Although somewhat hierarchical, semantics differ from the hierarchical decomposition approach in that the reactive subsystem is guided rather than controlled by the deliberative subsystem. Hence, the reactive subsystem, which runs continuously, parallel to, and independent from the deliberative subsystem, can respond quickly to outside events (such as approaching obstacles). A particular popular instantiation of hybrid architectures are the three layer architectures. One difficulty with hybrid architectures is the coordination of reactive and purposeful high-level behaviours.

Three Layer Architectures seem to be the current state of evolution. Three Layer Architectures usually employ three levels of abstraction (occasionally a fourth layer is added for servo control (Payton 1986)). These three layers are the deliberative layer, a sequencing layer, and a reactive layer. (Figure)

The deliberative layer uses classical AI representation and reasoning techniques such as (temporal) planning, scheduling, and resource handling. Activities on this layer correspond to long term strategic planning as well as eventual plan adapta tions. This level relies on very abstracted knowledge, highly sophisticated reasoning techniques, and is the typical domain of AI planners able to make use of extensive application domain knowledge. Planners used in this layer are IxTeT (Ghallab & Laruelle 1994), SIPE, and SIPE2 (Wilkins et al. 1994).

The sequencing layer involves a reactive planner (see above) which selects appropriate tactics using context dependent rules. A tactic is a pre-written partially ordered set of actions (behaviours, operators) called a task net. Task nets represent procedural knowledge and are rich in structure, i.e. disjunction and conjunction of action, recursion, and hierarchies of actions. The sequence layer selects appropriate task nets and executes them according to the precedence relationships within the task nets. Execution of task nets (which may involve recursive calls) leads to activation and termination of reactive layer behaviours. via monitoring, and recognition of termination conditions, and execution failures.

The reactive level then performs the transition from symbolic reasoning to non-symbolic numerical control and the combination of separate behaviours. Several techniques have been applied. Among these are a subsumption-like mechanism, blended behaviours, and “potential field” methods (Latombe 1991). Within three layer architectures, communication and interaction is of greatest concern. The general approaches are:

- **Deliberative layer - sequencing layer:** The connection between the deliberative and sequencing layer is done by mapping task nets to operator descriptions. Sequencing layer task nets are represented by planning operators in the deliberative layer. These planning operators are decomposed by task nets. Parameters are bound by unification.

- **Sequencing layer - reactive layer:** Interaction between these two layers involve (a) activation, (b) deactivation or termination, (c) passing of control parameters, and (d) monitoring of success. A more sophisticated approach (Saffioti 1993a) involves frame-like structures for objects in the world and behaviours to anchor internal symbols to external objects. Within the blended behaviour approach the composition of behaviour in the sequencing layer (represented as operators within task net structure) directly corresponds to the (fuzzy) composition of behaviours on the reactive layer. Anchoring behaviours are triggered automatically.

Recent research concentrates on the inidividual layers, the interfaces, and the communication flows between the layers. Two systems, the RAPs and the PRS, seem to emerge as general sequencing layer plug-ins and have been used within different three layer architectures together with different front and back ends (deliberative and reactive layer).

3.5 THREE LAYER ARCHITECTURE BACK AND FRONT ENDS

In this section we review several approaches dedicated to the reactive (back end) and deliberative (front end) levels of abstraction of three layer architectures. In order to compare different approaches we group them according to the place they would take in a three layer architectures. Since the differences among hybrid architectures are small and largely traced back to the choices made for implementing the individual layers we do not consider them here for comparison. For completeness we list some of the most influential hybrid architectures together with their applications in the following section.

3.5.1 Reactive Layer Approaches

The reactive layer with direct access to the robot effectors stays in closest contact with the environment. Complex control on this level involves reflexive and reactive responses such as object avoidance, corridor following, target tracking, object approaching, grasping, etc. characterised by low bandwidth high frequency closed loop interaction with the environment. Fast (bounded time) response based on low assimilated data is an absolute "must" on this level.

Although it is arguable whether the compiled plans approaches (Universal Plans, Teleo-reactive Trees, Situated Automatons) belong in this group their immediate, nonreflective nature of action selection makes them fit better into this group than into "sequencing layer approaches". Since the output of these control approaches is one or more unparametrised actions, i.e. activation/deactivation signals, a separate "servo level", as in the SSS architecture by Connell (1992), should be assumed to provide analog output to the hardware.

Blended behaviours, situated automatons, and universal plans have already been discussed, which leaves ALFA, Arkin's Motor Schemas and Teleo-reactive trees left for a brief description.

ALFA (Gat et al. 1991) is a language based on the subsumption paradigm and is used to specify and compile subsumption behaviours. The major difference from pure subsumption (Brooks 1989) is the use of arbitrary message formats between the state machines and the possibility of using internal states (though it is not encouraged).

Arkin's Motor Schemas (Arkin 1989) are included since they are unique in using "potential fields" for composing complex behaviours from simple ones. A major step forward from this is the blended behaviours approach.

Teleo-reactive Trees (Nilsson 1994) are a relatively new technique for encoding behaviours. Teleo-reactive trees are sets of condition → action rules which are continuously evaluated. The first action which evaluates to true is executed and continues as long as the energising conditions are true. Teleo-reactive trees can be recursive and organised in hierarchies. Although the character of the teleo-reactive approach is more that of a sequencing machine (like RAP or PRS) since primitive actions are either on or off (no parallel actions in its original form, no analog control), the continuous nature of actions and the continuous evaluation of applicability (called energising conditions) puts them closer to low layer approaches than e.g. RAP or PRS. The major advantage of teleo-reactive trees is easy implementation: hardly any framework is needed for this. The major disadvantage is inefficiency since all energising conditions must be evaluated continuously.

Below we compare key aspects of reactive approaches. Table 4 compares additional properties.

Type of Approach:

ALFA (Gat 1991): subsumption variant, circuit semantics.
Blended Behaviours (Saffioti et al. 1993b): fuzzy rules over a local perceptual space (LPS).
Arkin's Motor Schemas (Arkin 1989): biological motivated motor control schemas.
Situated Automaton Theory (Kaelbling & Rosenschein 1989): situated automaton theory.
Subsumption (Brooks 1989): interconnected augmented finite state machines (AFSM), circuit semantics.
Teleo-reactive Trees (Nilsson 1994, Benson & Nilsson 1995): production rules with circuit semantics, compiled plans.
Universal Plans (Schoppers 1987): compiled plans as decision trees for a fixed set of tasks mapping the world state into the next action to take.

Smallest Computational Entities:

ALFA: state-machine and dataflow semantics, analog and digital transfer functions.
Blended Behaviours: fuzzy rule sets.
Arkin's Motor Schemas: "motor schemas" or arbitrary complex transfer function.
Situated Automaton Theory: situated automatons.
Subsumption: augmented finite state machines.
Teleo-reactive Trees: teleo-reactive rules, condition - action pairs.
Universal Plans: situation - action pairs.

Reference To Specific External Objects:

ALFA: no.
Blended Behaviours: yes, through anchoring.
Arkin's Motor Schemas: implicit via perception schemas.
Situated Automaton Theory: yes, but anchoring is assumed to be provided by the sensing system.
Subsumption: no.
Teleo-reactive Trees: not really, however parameter binding exists.
Universal Plans: no, assumed to be part of the sensing subsystem.

Composition of Complex Behaviours:

ALFA: as wired.
Blended Behaviours: disjunction, conjunction, chaining defined by fuzzy logic.
Arkin's Motor Schemas: merging via the potential field by summing up individual MS responses.
Situated Automaton Theory: defined in specification. Conflicting behaviours are not possible if compile time specification is ok.
Subsumption: as wired.
Teleo-reactive Trees: no.
Universal Plans: situation dependent. Several actions could be invoked resulting in some form of complex behaviour.

Interactions of plan components within the same layer:

ALFA: suppression, inhibition, arbitrary signals and messages through channels.
Blended Behaviours: composition. Behaviour arbitration through context dependent fuzzy meta-rules. Centroid defuzzification suggested to avoid conflicts among active behaviours.
Arkin's Motor Schemas:fusion via the potential field.
Situated Automaton Theory: as wired.
Subsumption: suppression, inhibition, fixed length binary vectors through channels.
Teleo-reactive Trees: none.
Universal Plans: none.

Interaction with the next higher layer in hybrid architectures:

ALFA: activation, deactivation.
Blended Behaviours: symbolically through symbol grounding via the LPS.
Arkin's Motor Schemas: instantiation of MS, status reports from MSs.
Situated Automaton Theory: activation, deactivation.
Subsumption: activation, deactivation.
Teleo-reactive Trees: activation, deactivation.
Universal Plans: not implemented in a hybrid architecture (activation, deactivation).

Implementation Status and Support:

ALFA: Language support “ALFA”.
Blended Behaviours: applied technology (see below).
Arkin’s Motor Schemas:applied technology (see below).
Situated Automaton Theory: tool support (GAPPS, REX, RULER).
Subsumption: Language support “Behaviour Language”.
Teleo-reactive Trees: applied in simulated environments.
Universal Plans: interpreter for universal plans exists.

3.5.2 Deliberative Layer Approaches

The deliberative layer is the only layer which maintains a symbolic world model. Representative applied planners include IxTeT and SIPE/SIPE2.

- **IxTeT** (Ghallab & Laruelle 1994) is a hierarchical, temporal, least commitment, partial order planner with parallelism which has been used in the deliberative layer of a control system for (simulated) Mars robot missions. It is used

Table 4. Reactive Layer Approaches.

Criterium	*ALFA*	*Blended Beha-viours*	*Arkin's Motor Sche-mas*	*Situated Automa-ton Theory*	*Sub-sumptio n*	*Teleo-reactive trees*	*Universal Plans*
type of out-put vector	analog	analog	analog	binary	binary	binary	binary
sensing, perception	yes	yes	yes	no	yes	no	no
system de-velopment	hand-crafted	hand-crafted	hand-crafted	run time system[1]	hand-crafted	hardco-ded rules[2]	run time system[3]
internal states	yes	yes	yes	yes	no	yes	yes
time bounded response	yes	not guar-anteed	not guar-anteed	yes, guar-anteed	yes	not guar-anteed	yes
hierarchies	layers of compe-tences	no	no	no	layers of compe-tences	de-compo-sition	no

(1) Compiled from abstract description using goal reduction and goal regression rules.
(2) Learning is currently explored (Benson & Nilsson 1995).
(3) Compilation of UPs through goal regression. Use of abstract domain description and goal description.

in connection with PRS.

- **SIPE/SIPE2** (Wilkins et al. 1994), (Wilkins & Myers 1995) is similar to IxTeT and incorporates temporal reasoning, parallel action, and operator hierarchies as well. Special emphasis, as with IxTeT has been put on efficient heuristics guiding the search process during planning.

3.6 APPLIED TECHNOLOGIES

In the following, a few architectures together with some of their reported applications are listed.

Hierachical Architectures:

NASREM / RCS, reference architecture for hierarchical system decomposition: Subsets of the proposal have been implemented in: Horizontal Machining Workstation, Cleaning and Deburring Workstation, Advanced Deburring and Chamfering System, Space Station Telerobotic Servicer, Coal Mining Automation architecture, Nuclear Submarine Maneuvering System, Control System for Multiple Autonomous Undersea Vehicles, System for Remote Driving, Control System for a U.S. Postal Service Automated Stamp Distribution Center, and an Open Architecture Enhanced Machine Controller. (Albus et al. 1989), (Albus 1995).

Subsumption Architectures:

Pure Subsumption, Behavior Language: Mobile robots (simple behaviors: force balancing, obstacle avoidance, wandering, following, gait control) (Brooks 1989)

ALFA: Subsumption with extended circuit semantics (Gat 1991). ALFA (Gat et al. 1993) is used in ATLANTIS (see below).

Sequencing Layer Approaches:

Procedural Reasoning System (PRS): Several implementations of PRS have been carried out, e.g.: SRI-PRS, PRS-CL, C-PRS, UM PRS. PRS-concepts are used in the following applications: Reaction Control System of NASA´s space shuttle, Interactive Real-Time Telecommunications Network Management System (IRTNMS), (Georgeff & Lansky 1987), (Ingrand et al. 1995), (Wilkins et al. 1994), (ACS 1995) (also in the FLAKEY architecture and within the LAAS architecture. See below)

Reactive Action Packages (RAPs): RAP is used within the following applications: Autonomous vacuum cleaner (ongoing project), also used in the sequencing layer in hybrid architectures (see below). (Firby 1995)

Situated Automaton Theory: A number of tools exist which support situated automaton theory, e.g.: GAPPS, RULER, REX: Mobile robot navigation (Kaelbling & Rosenschein 1989).

Teleo-reactive trees: These have so far only be applied within simulated environment, i.e. Botworld (simulation) (Nilsson 1994)

Reactive Layer Approaches:

Blended Behaviors are used in the FLAKEY architecture. (Saffioti 1993a)

Bonasso's architecture is based on situated automaton theory. Applications include mobile robots for navigation, retrieval, delivery, and reconnaissance tasks indoors and outdoors as well as undersea operation. (Bonasso 1991)

Deliberative Layer Approaches:

SIPE and *SIPE2* are partial order temporal planner and are used in various applications high-level symbolic temporal reasoning and planning, e.g. managing aircraft on carrier decks, travel planning, construction tasks, and mobile robots

AP is a SIPE-like, state based, partial order, hierarchical planner. It is used in 3T (see below).

IxTeT is also a partial order temporal planner. It is used in the LAAS architecture (Chatila et al. 1992).

Other Hybrid Architectures:

Touring Machine have layers which operate independently and in parallel (no hierarchy). Action arbitration is performed by a separate control system. (Ferguson 1992). Touring Machine have been investigated in a simulated 2D multi-agent environment for evaluating spatio-temporal reasoning in partially-structured dynamic real-time environments. (Touring World) (Ferguson 1995)

Three Layer Architectures:

3T includes a SIPE-like planner and a RAP-like sequencer. 3T is a successor of the Bonasso architecture (Bonasso 1991, Bonasso & Kortenkamp 1994). It is used for mobile land and undersea robots for defense applications. Mobile robots for navigation tasks, following, approaching, and obstacle avoidance. Mobile armed manipulator system for manipulation tasks: ARMSS (Automatic Robotic Mainte nance of Space Station) (dual-armed). EVA Helper/Retriever for maintenance tasks around a space station (simulation thereof). (Bonasso & Kortenkamp 1994, 1995)

ATLANTIS includes a simulation based planner, a RAP-like sequencer (extended by resource handling and mutual exclusion of certain behaviors using a semaphore system), and ALFA. ATLANTIS is used in mobile robots for indoor and outdoor navigation (JPL Mars rover testbed). Additional tasks include collect and delivery tasks and refuelling. (Gat 1991, 1993)

The *FLAKEY architecture* includes PRS and blended behaviors. Mobile robots (complex indoor navigation tasks). (Saffioti et al. 1993b)

GLAIR includes a Sensori-Actuator Level (low-level behaviors and reflexes), Perceptuo-Motor Level (model free high-level behaviors) and a knowledge level (BDI level). Air Battle Simulation, mobile robots (simple search and follow tasks). (Hexmoor et al. 1993), (Hexmoor 1995)

The *LAAS architecture,* includes IxTeT (see above) and a sequencing layer based on finite state automatons (later replaced by PRS). It is used for mobile robot navigation, planetary exploration, mobile robot navigation in outdoor environments (ADAM), and controlling a fleet of autonomous mobile robots to handle container transportation (ESPRIT MARTHA).(Chatila et al. 1992), (Ingrand et al. 1995)

The *Payton architecture* includes 4 layers. The bottom 2 roughly correspond to the reactive layer in three layer architectures. It is used for navigation in a simulated environment. (Payton 1986)

SSS includes three layers: A symbolic (roughly corresponding to the sequencing layer), a subsumption, and servo layer. No true deliberative layer is included. Application is indoor navigation for a mobile robot (Connell 1992).

Other Architectures:

Autonomous Robot Architecture (AuRA) is a collection of five subsystems: perception, cartographer, planner, motor control, and homeostatic control. (Arkin 1989)

PEM is used within the control system of an autonomous paper roll manipulator. (Heikkilä & Röning 1992)

The *Task Control Architecture (TCA)* is used within several mobile robots, e.g. Ambler (six legged robot for planetary exploration), Ratler (semi-autonomous lunar missions), and Xavier (indoor navigations). (Simmons 1990, 1995)

3.7 DISCUSSION

In this chapter we discussed the different major architectures in the field of robot control together with the dimensions of applications. It became apparent that many of the discussed techniques and robot control architectures have their own, more or less vaguely described, field of application. No generally suited architecture can be devised.

Several architectures follow a three layer approach in which the top most layer exhibits deliberative reasoning, the middle layer template based planning, and the lowest level reactive high frequency interaction with the environment by means of subsumption like behaviours, fuzzy reasoning, or traditional servo loops. These architectures cover the whole bandwidth of reasoning. Engineering aspects, however, of how to create such control systems in a systematic way are largely undiscussed. The organisation of knowledge and the flow of information and control between the layers are still open questions.

The task control architecture (TCA) emphasizes structured control as the key element of combining logically and physically distributed control entities (such as sensors and actuators) and provides a generic framework for building intelligent distributed control systems. Our architecture, presented later in this thesis, shares much of this motivation. It adds and augments several features which to some extent are neglected in many of the discussed architectures. These include monitoring plan execution and validity, intention switching, and fault recovery.

4 THE ARCHITECTURE

In this chapter we propose and describe a new robot control architecture. The objectives of this architecture are:

- **Planning** to provide context dependent intelligent behaviour
- **Concurrency** to allow parallel activities
- **Monitoring** to detect errors in an early stage
- **Intention Switching** to swap tasks for more critical/urgent ones
- **Fault Recovery** to employ minimum invasive strategies to recover from errors
- **Integration** of the above in a robot control architecture

An overview of the system is given in Figure . The major components of our architecture are the *plan components* and the *current execution graph* which connects them.

- *Plan Components* are generic activities. We distinguish among virtual plan components responsible for monitoring conditions; abstract plan components, which are high level tasks that are successively decomposed into less abstract activities; and atomic plan components, which are not further decomposed but

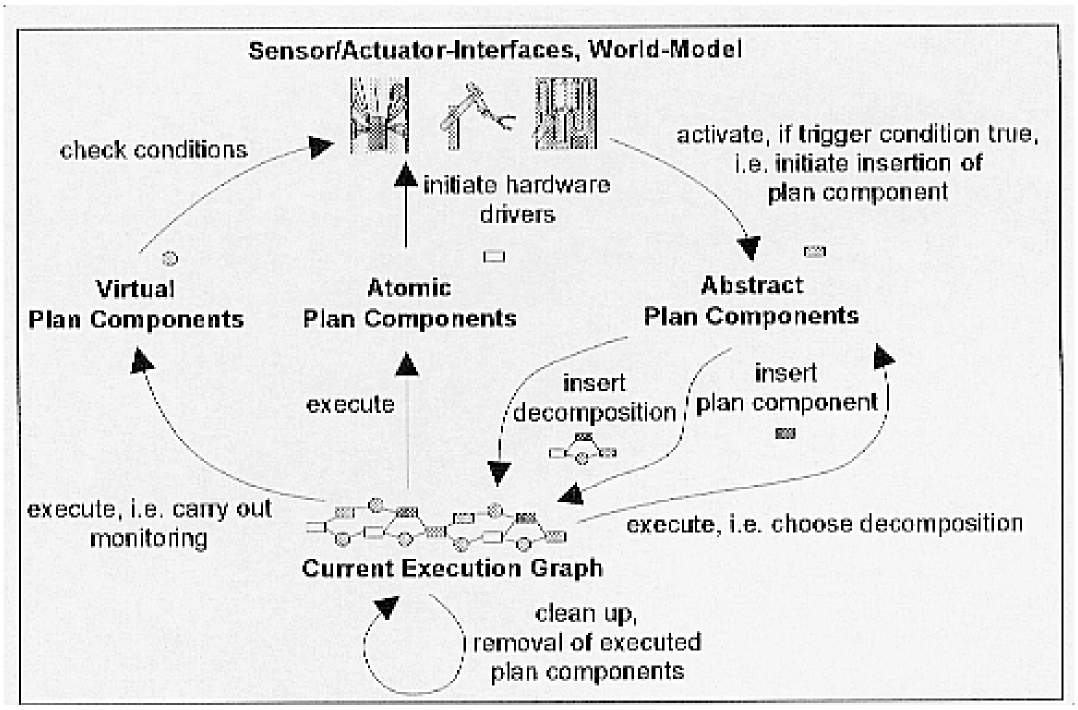

Sensor/Actuator-Interfaces, World-Model
check conditions
initiate hardware drivers
activate, if trigger condition true, i.e. initiate insertion of plan component
Virtual Plan Components
Atomic Plan Components
Abstract Plan Components
execute
insert decomposition
insert plan component
execute, i.e. carry out monitoring
execute, i.e. choose decomposition
Current Execution Graph
clean up, removal of executed plan components

Figure 21. Architecture Overview.

are directly tied to hardware drivers.

Plan components are inserted into the current execution graph as a result of decomposing an already existing (abstract) plan component in the current execution graph (decomposition) or as a response to an abstract plan component's trigger conditions (intention switching). This latter case enables the robot control architecture to respond to critical environmental conditions by insertion of an abstract plan component into the current execution graph followed by the decomposition of that plan component.

- The *current execution graph* keeps track of the planned activities and their precedence relationships, initiates execution of plan components, and removes executed plan components. The current execution graph also determines whether newly inserted plan components or decompositions are in conflict with already existing plan components in the current execution graph.

4.1 INTRODUCTION

We follow the common view of actions as the generic entities of intelligent control in discrete event systems (Currie & Tate 1991), (Fikes & Nilsson 1971), (Firby 1987), (Heikkilä & Röning 1992). Actions are typically used to represent both physical activities such as actuator movements and higher level abstract tasks. In this context, actions can be treated as STRIPS operators (Fikes & Nilsson 1971). We extend this notion towards plan components to include virtual actions, i.e. internal activities such as monitoring request related to certain subgoal-producer-consumer relationships such as causal links (Penberthy & Weld 1992).

Within this thesis "plan components" include abstract higher level activities,

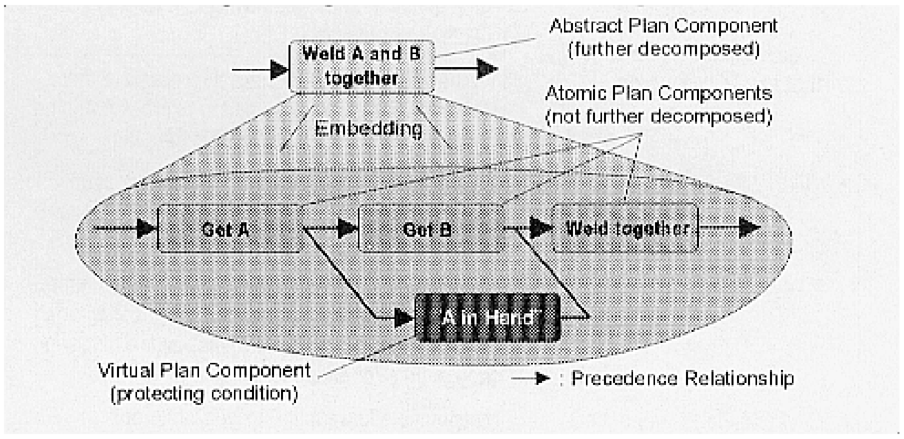


Figure 22. Different Types of Plan Components.

lower level physical activities, and virtual plan components expressing dependencies among plan components. Virtual plan components are used to monitor plan execution and plan validity.

Three types of virtual plan components are employed for this purpose. Those which check initial states (I-virtual plan components), transient states (T-virtual plan components), and those which check goal states (G-virtual plan components). Table 5 describes the proposed taxonomy of plan components. Virtual plan components are constructed to explicitly represent certain monitoring tasks, such as monitoring producer-consumer relationships (Figure and Figure). Virtual plan components have two major uses:

- Virtual plan components are used to express the conditions underlying the Modal Truth Criterion. Once these relationships are determined at design time the conditions to be checked are readily available for monitoring and do not

Table 5. Plan Component Type Taxonomy.

Plan Component Type	***Description***
Physical Plan Component	Plan components which are directly or indirectly related to a physical activity.
	Plan components can be defined as reactive in order to react to certain trigger conditions. Reactive plan components are used to model high-level behaviours.
Abstract Plan Component	Conceptual plan components which are later further decomposed by more detailed subplans.
Atomic Plan Component	Atomic plan components which are directly connected to hardware drivers.
Virtual Plan Component	Virtual plan components are not related to any physical activity but are used to express planning and execution constraints within the plan.
I-Virtual Plan Component	responsible for monitoring *initial* state descriptions, used to check the existence of certain initial-conditions
T-Virtual Plan Component	responsible for monitoring *transient* state descriptions, also used to monitor persisting conditions (subgoals, planning constraints) over a longer period of time (continuous checking).
G-Virtual Plan Component	responsible for monitoring *goal* state descriptions, i.e. used to check the existence of certain goal-conditions.

need to be recomputed.

- Virtual plan components may represent protection intervals i.e. dynamic planning constraints set up by other agents or as subplan constraints.

Virtual plan components are requests for monitoring and correspond to the conditions listed in STRIPS operator’s precondition, add, and delete lists. Transient states are described using T-Virtual Plan Components which can be used to represent postcondition-precondition relationships.

Physical plan components correspond to physical activities whereas virtual plan components correspond to monitoring requests. This current view leaves space for alternative interpretations and future research. Knowledge achieving tasks (active sensing) could result from the decomposition of virtual plan components. Computational tasks which are reasonably represented as physical (atomic) plan components may not result in any physical activity. The key concepts introduced for physical plan components are:

- **Operator Models**: The representation of plan components extends the STRIPS formalism (Fikes & Nilsson 1971; Hendler & Subrahmanian 1990, pp. 63 - 65) by adding transient state information, plan component states, and temporal information. This information is automatically set up during the instantiation process. Plan component states indicate the current state of the plan component within its life cycle, e.g. whether it is just planned or already scheduled for execution. Employing states and state transitions enables the control system to easily keep track of the execution process and the necessary monitoring task for each plan component state.

- **Hierarchies**: Abstract plan components can be decomposed by lower level subplans, producing hierarchical plans. This process is called operator (action) abstraction (Sacerdoti 1974) and hierarchical task reduction (Hendler & Subrahmanian 1990).

- **Parametrisation**: During decomposition, when instantiating plan components, planning parameters must be bound. For example, when decomposing a transport task with a pick-move-place sequence, target and destination coordinates must be determined to instantiate “pick”, “place”, and “move” plan components.

- **Duality of planning and execution**: Planning results from executing abstract plan components, whereas physical execution results from execution of physical plan components.

Plan components result from instantiation of plan component classes. The key properties of plan component classes are:

- **Object Orientation**: Physical plan components are objects in the sense that they employ a common interface and interact with other parts of the system through well defined messages. The content of these messages is normal control information, such as *start execution* or *execution finished*; exceptional control information, such as notifications about erroneous situations; or planning constraints. Such constraints either affect plan creation, e.g. "give me a plan within 10 seconds", or constrain the resulting plans, e.g. "give me a plan for assembling 'A' lasting at most 10 seconds and using only robot 'X'". Physical plan components closely resemble the agent-like objects proposed by Shoham (1992).

- **Black Box Character**: Physical plan components are black boxes. The way their computational engine works is hidden. Depending on their actual nature (assembly task, grasping, navigation, object recognition, etc.) specialised techniques, such as fuzzy control for servo control, or neural networks for object recognition, or HTN planning for task decomposition, may provide the "intelligence" for planning (decomposition) and execution (control).

- **"self-initiation":** Physical plan components can be defined to respond to particular trigger conditions. These reactive abstract plan components become critical under certain trigger conditions and require immediate or rapid attention for intention switching, decomposition, and execution. Reactive physical plan components (high-level behaviours) allow the designer to keep conceptually different tasks, e.g. hovering and recharging batteries, separated within the control system.

4.2 PLANS

Plans are the central data structure within planners and control systems for intelligent robots since they define the sequence and alternatives of goals that are attempted. Traditional plans were linear, i.e., totally ordered like STRIPS plans, or non-linear, i.e., partially ordered like TWEAK plans. Recently plan structures have been proposed which include control structures including choice, loops, and abstraction. Hence, plans more and more resemble programs. The introduction of control structures increases compactness, expressiveness, flexibility, and readability but also increases the computational burden to derive plans from scratch.

Improved readability implies improved writability, which is important if plans are designed by people. Although we recognise the advantages of high-level control structures, we stay with a rather simple, though powerful, mechanism in order to include functionalities such as intention switching which would be quite difficult, though not impossible, to introduce using complex control structures.

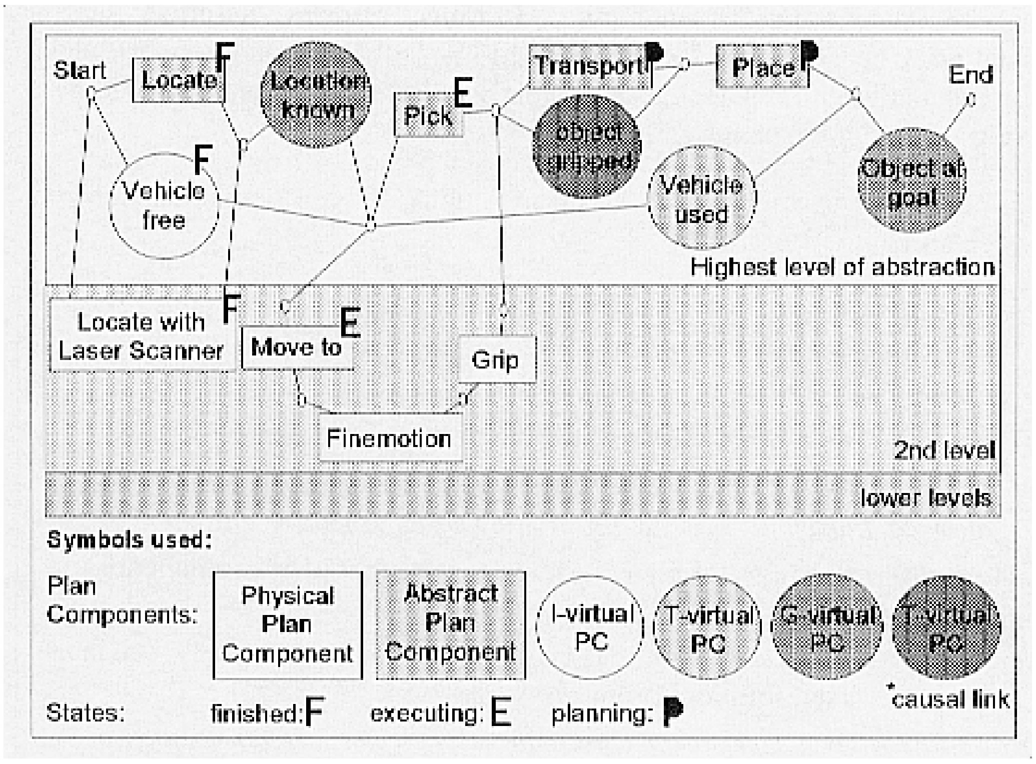


Figure 23. A Plan.

In the proposed robot control architecture, plans are used for two purposes (a) to describe the task nets for decomposing abstract plan components, i.e. the structure of possible instantiations, and (b) as the representation of the currently planned activities within the current execution graph. The current execution graph shown in Figure depicts a snapshot of a pick-transport-place sequence.

Within this simplified current execution graph, virtual plan components denote various conditions, such as (a) the result of executing a plan component (postcondition), e.g. “location known” as a result of “locate” or “object gripped” as a result of “pick”, (b) constraints which hold during the execution of a plan component, e.g. “vehicle used” denoting the use of a resource “vehicle” during execution of “transport” (transient state description), or (c) certain preconditions which are prerequisites to the execution of plan component, such as “vehicle free” for “pick”.

Postconditions that are simultaneously preconditions of another plan component form causal links or postcondition-precondition relationships are represented as T-virtual plan components. T-virtual plan components monitor a prevailing condition, e.g. transient states or causal links, as long as they are executed.

T-virtual plan components are initiated upon initialisation of the related physical or abstract plan component and are terminated when the related plan component finishes. In Figure “object gripped” describes such a situation. The condition “object gripped” is both postcondition of “pick” and precondition of “place”. Consequently, this condition is monitored by a T-virtual plan component.

Plan components are defined as instantiations of plan component classes which are defined as:

Definition 2. (Plan Component Class) A plan component class is a system *plan_component_class = (pc_type , pc_allowed_parameters , pc_expansions, conditions , instantiation)* :

pc_type : specifies the type of the plan component class; *pc_type* $\in$ *{abstract, atomic, t-virtual, i-virtual, g-virtual}*

pc_allowed_parameters: is the set of allowed parameter vectors.

pc_expansions: is a set of plan expansions. (abstract plan components only, see later)

conditions: is a set of conditions (virtual plan components only, see later) used for monitoring.

instantiation: *plan_component_class*parameter_vector* $\rightarrow$ *plan_component* the instantiation function for the plan component class. It is defined for *parameter_vector* $\in$ *pc_allowed_parameters*.

Plan components can then be defined as:

Definition 3. Plan Component. A plan component is a n-tuple: *plan_component = (class , parameters , m_conds , status):*

class: is the plan component class of the plan component.

parameters: is a vector of instantiation parameters of the plan component; *parameters* $\in$ *pc_allowed_parameters* of plan component class “*class*” .

m-conds: a set of condition - parameter_vector pairs. Used for monitoring. Resulting during instantiation by parametrisation of plan component class conditions.

status $\in$ *{planning, executing, executed }*; execution, planning status.

The set of all plan components is the plan component universe. The plan component universe consists of two distinct sets of plan components: Terminal plan components, i.e. virtual and atomic plan components, and abstract plan components, which can be decomposed.

Definition 4. Plan Component Universe. The plan component universe PC is the set of all possible plan components. PC consists of atomic plan components TPC, i.e. terminal non-decomposable plan components, and abstract plan components APC which are further decomposed. It holds:

$PC = APC \cup TPC$, *with* $APC \cap TPC = \emptyset$

$\forall pc \in APC \Leftrightarrow c = class_{pc} \wedge pc\_type_c = \{abstract\}$

$\forall pc \in TPC \Leftrightarrow c = class_{pc} \wedge pc\_type_c = \{atomic, t\text{-}virtual, i\text{-}virtual, g\text{-}virtual\}$

Plans consist of plan components and are represented as *doubly pointed graphs*, i.e. as directed graphs with edges denoting plan components and vertices connecting edges. Moreover, each plan has one “begin” and one “end”-node. See below for a formal definition.

Definition 5. Doubly Pointed Graphs / Plans. A double pointed graph over an alphabet $PC=APC \cup TPC$ of terminal (*TPC*; atomic/virtual plan components) and non-terminal (*APC*; abstract plan components) plan components is a system $H = (Nodes, Edges, label, from, to, begin, end)$ with:

Nodes = a set of nodes, i.e. time points

Edges = a set of edges, i.e. plan components

$label : Edges \rightarrow PC$ = an edge labelling function, i.e. plan component

$from : Edges \rightarrow Nodes$ = the source node (from-node of an edge)

$to : Edges \rightarrow Nodes$ = the target node (to-node of an edge)

$begin \in Nodes$ = the start node of the graph

$end \in Nodes$ = the end node of the graph, with begin≠end

$from \neq$ to are such that
no loops exist
all nodes and edges are connected
$\forall edge \in Edges\ \ from(edge) \neq end$
$\forall edge \in Edges\ \ to(edge) \neq begin$
$\forall node \in Nodes\ \ v \neq begin \Rightarrow \exists\, edge \in Edges\ \ from(edge) = node$
$\forall node \in Nodes\ \ v \neq end \Rightarrow \exists\, edge \in Edges\ \ to(edge) = node$

Within an implementation plans could be defined as (C++):

```
class plan
   {
   plan_component_list     plan_components;
   node_list               nodes;
   ordering_list           plan_component_ordering;
   ...
   }
```

The `ordering_list` is a list of plan component orderings, e.g. pairs of before and after nodes.

4.3 CURRENT EXECUTION GRAPH

The current execution graph is a special instantiation of a plan, the one which holds the currently planned activities.

After instantiation, i.e. decomposition, of an abstract plan component, a new sub-plan is generated and inserted into the current execution graph. The current execution graph is a central data structure holding information about all planned activities, constraints, and the execution status (life-cycle status) of all plan components.

Operations on the current execution graph are mutually exclusive with respect to writing (insertion and deletion) in order to ensure consistent transactions.

Plan components are executed and decomposed in chronological order, although decomposition of later plan components could be sometimes carried out at earlier stages. This is a point to be further investigated.

4.4 PLAN COMPONENT CLASSES

Plan component classes are static objects providing knowledge about

- how to decompose abstract plan components into task nets of simpler components
- how to execute virtual and atomic plan components

Plan component classes are used to hide domain specific planning algorithms, such as task, assembly, or path planning.

Using C++, plan component classes are defined as:

```
class plan_component
 {
 plan_component_type             type;
     // one of {abstract, atomic, I, T, G}
 plan_component_name             name;
     // name, reference, and monitoring condition
 plan _component_parameters   args;
 life_cycle_status               state;
     // one of {“planned”, “executing”, “finished”}
 result_type                     fnc(...);
     // the decomposition, execution, monitoring routine
 fnc_duration_frequency       duration;
     // frequency and approximate duration of execution
 ...
 }
```

`result_type` returns the result after applying `fnc` (e.g. the instantiation function in case of abstract plan components). The result may be an error, an “ok”, or a decomposition (e.g. a subplan) to be inserted into the current execution graph.

4.5 PLAN COMPONENTS

Plan components are dynamic objects created by instantiation of plan component classes by applying plan component parameters (if any) to the instantiation functions of abstract plan component classes.

As a result of planning a task net is produced, if possible. Task nets consist of physical and virtual plan components. All plan components have in common:

- Parent plan component (unless they are top-level abstract plan components).
- Instantiation parameters.
- Information about duration and frequency of execution.
- Plan component life-cycle state. Plan components are dynamic in that they exhibit a limited life time between instantiation and deinstantiation, when the execution of the plan component has successfully or unsuccessfully finished. The plan component life cycle state specifies the plan components within the plan components life cycle.
- A reference to its plan component class in order to allow access to procedural knowledge, such as driver routines (for atomic plan components and virtual

plan components) and instantiation functions (for further decompositions abstract plan components).

T-virtual plan components include a reference to the physical plan component to which they are related (if any). Virtual plan component include a description of expected conditions or behaviours. The constraints to be surveyed by the monitoring system and to be checked during intention switching are modelled by a description language. The description language is a collection of predicates which describe states and state changes. Since the description language is very application dependent, only the necessary properties of the description language are described here.

Aside from symbolic description of states, e.g. on(A,B), modelling approaches such as QSIM (Kuipers 1986) are suitable for the continuous event parts of the system. Description languages provide a way to describe discrete and continuous states and changes. To relate behaviour descriptions to each other, e.g. to determining whether they conflict or not, we introduce monitoring predicates.

Four monitoring predicates must be defined for all propositions of the description language. The monitoring predicates are *matching, not_matching, consistent, not_consistent.* For the sake of simplicity they evaluate to *true* or *false.* However monitoring predicates may also be evaluated using other representations of belief if deemed necessary.

Definition 6. Monitoring Predicates relate conditions (m_conds) as part of (virtual) plan components to each other and sensor readings in order to facilitate monitoring. *m* and *n* are propositions (conditions). *s* denotes a sensor reading.

matching(m,s) $\rightarrow$ *{True, False}*

A proposition m (of the description language) matches with given sensor readings *s*, or not.

not_matching(m,s) $\rightarrow$ *{True, False}*

A proposition m does not agree with given sensor readings s, or not.

consistent(m,n) $\rightarrow$ *{True, False}*

Two propositions are consistent, if the first one implies the second one, i.e. $m \rightarrow n$, e.g. *consistent(x > 23, x > 12) = True*. The following does not necessarily hold: *consistent(m,n)* $\Leftrightarrow$ *consistent(n,m)*

not_consistent(m,n) $\rightarrow$ *{True, False}*

Two propositions are not consistent, if the second one does not imply the first one, i.e. $m \rightarrow \neg n$. The following does not hold:

consistent(m,n) $\Leftrightarrow$ $\neg$*not_consistent(m,n)*

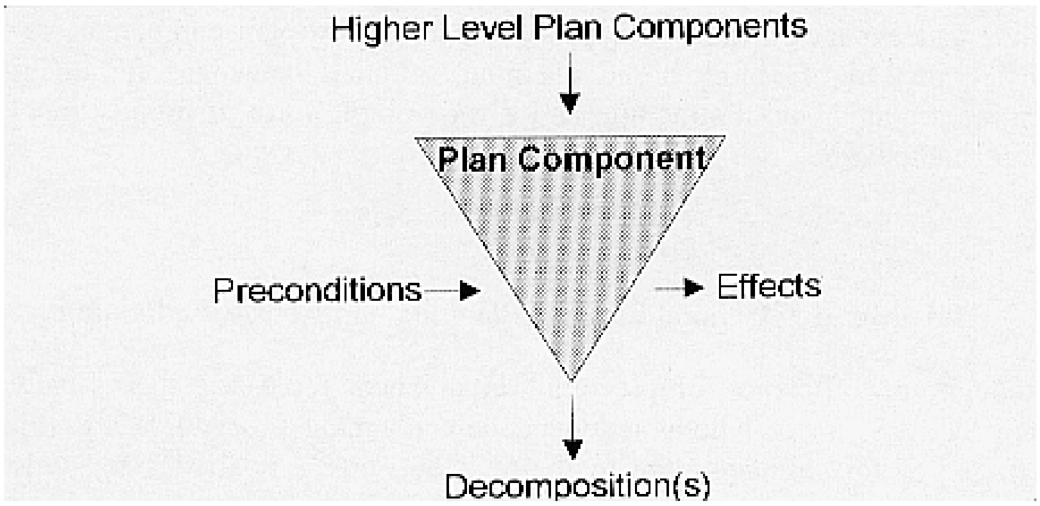


Figure 24. Plan Components.

Decomposing an abstract plan component expressing an arm movement from a location A to a location B may yield a subplan consisting of a number of virtual plan components monitoring the precondition, the transient states, and the results of executing one atomic plan component "MoveArm(A, B)". Figure illustrates this decomposition with "begin" and "end" nodes omitted.

4.6 PLAN DYNAMICS

Plans are dynamic structures which are modified by plan component instantiation, execution, and intention switching. These modifications are described below using graph grammars applied over doubly pointed graphs.

A graph grammar is a collection of graph rewriting rules which, applied to an initial graph, forms the language (e.g. the set of all possible plans in our case) of the graph grammar. We apply graph grammars and rewriting in different ways to carry out plan modification:

- **Plan component instantiation**: The actual planning process with the traditional meaning. (Chapter 4.7)
- **Plan component deinstantiation:** Removing plan components from the plan. This may be done after execution, errors, or replanning. (Chapter 4.8)

- **Intention Switching:** insertion and decomposition plan components. (Chapter 4.9)

The complexity and dynamics of planning is usually addressed by applying skeletal, hierarchical, or related planning approaches (Puppe 1990, pp. 124 - 180), which imply the use of abstract actions, such as subplans or plan frames, which are not related to any physical action directly, but may be later refined and replaced by more precise plans.

4.7 PLAN COMPONENT INSTANTIATION

Plan components are created by instantiating *plan component classes*. Plan components are dynamic objects which become instantiated during decomposition.

Instantiation is carried out by an instantiation function, which is attached to each plan component class to derive a decomposition, i.e. a task net of subordinate plan components, which consists of physical plan components, associated I, T, and G-virtual plan components describing pre, post, and transient conditions, and further T-virtual plan component to describe precondition-postcondition relationships (causal links).

The plan component life cycle state is set to “newly instantiated”. Information about duration and frequency (e.g. for T-virtual plan components and atomic plan components) of execution may be added to each plan component if necessary and available.

During decomposition an abstract plan component decomposes itself, i.e. it adds a “subplan” to the current execution graph. The parent plan component is kept in the current execution graph to allow fallback after execution or in case of local fault recovery. Decomposition is shown in Figure . (parent plan component removed for clarity)

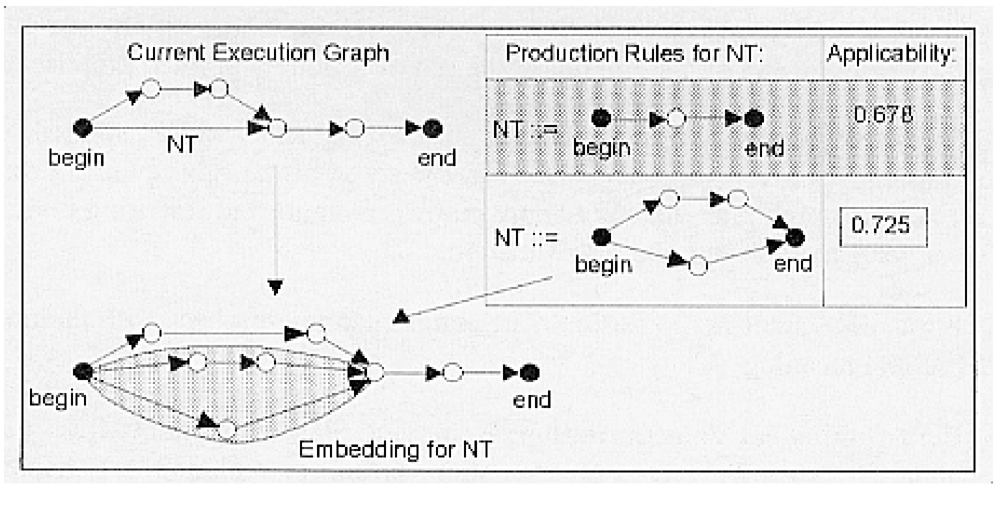


Figure 25. Decomposition of the Abstract Plan Component “NT”.

Although any kind of method to derive a decomposition may be used an HTN style decomposition (Erol et al. 1994a,b) schema may be suitable for most applications. The proposed decomposition mechanism uses predesigned plans (since the knowledge of how to perform an action is usually well known beforehand) and selects the most appropriate one for a limited set by means of (fuzzy) decision making.

Graph grammars are defined for plan component classes. The planning process is conceptually divided into selection of a plan component instantiation, plan expansion, and plan insertion. See Figure 3. Plan component instantiation includes the selecting the best applicable production rule. In most cases a simple situation dependent heuristic is sufficient for this. After having selected the best applicable rule, plan expansion preconditions must be checked to guarantee that plan variables used in the decomposition can be bounded based on available information. In case plan component decomposition does not succeed decomposition may be attempted with best applicable production rule not tried before (if any).

Plan Component Instantiation

- Choose the (remaining) plan expansion which has highest applicability of all applicable plan expansions.
- Check the plan expansion preconditions which are in addition to the plan component preconditions, if plan variables are used.

Plan Expansion Function

- Assign values to the plan variables if plan variables are used. This is called symbol grounding.
- Apply the plan expansion function to the bound variables.

Plan Insertion into the Current Execution Graph

- Plan expansion is compatible/consistent with an existing plan, if it does not destroy any subgoal preconditions nor does cause conflicts with actions concurrent to the plan component expansion.
- Structurally embed the plan expansion within the current execution graph (graph rewriting). Set the virtual actions according to producer-consumer-relationships caused by the newly inserted plan expansion.

Figure 3. Plan Component Decomposition.

The steps listed in Figure 3 can be carried out different orders. It is a matter of further study which sequence of checking these conditions is the most economical one.

Instantiation itself can be accomplished in very different ways. For simple task planning we propose a method based on graph grammars, more precisely using *context-free edge replacement rules.* Although the rules within the graph grammar framework are context dependent their selection is not. It is based on a fuzzy logic-based applicability control system, though any heuristic to select the best suitable decomposition rule may be used instead. We briefly describe this method below.

Plan expansions are *context-free edge replacement rules* (Definition 7; Habel et al. 1987). The left hand side of a plan expansion rewriting rule is plan component class and the right side is a *doubly pointed graph* without loops (partially ordered). Edges correspond to plan components and nodes correspond to points of execution similar to places in P/T-type Petri-nets. This representation, called the current execution graph, is used both as a snapshot of the current planning- and execution context and as the right hand side of plan expansion rules.

Definition 7. Context-free Edge Replacement Rule. A context-free edge replacement rule (*CFERR*) over an alphabet $PC=APC \cup TPC$ of nonterminal symbols *APC* (abstract plan components) and terminal symbols *TPC* (atomic/virtual plan components) is defined as (G_{PC} is the set of all doubly pointed graphs over PC):

CFERR = (lhs, rhs), with
$lhs \in APC$, left hand side
$rhs \in G_{PC}$, right hand side,

The set of all context-free edge replacement rules determines the planning universe, i.e. the language of the resulting *context-free edge replacement graph grammar* or the set of all possible plans (Definition 8). The language formed by the graph grammar may be infinite due to recursive graph rewriting rules. Using end recursive rewriting rules it is possible to model loops.

Definition 8. Context-free Edge Replacement Grammar.A context-free edge replacement grammar (*CFG*) over an alphabet $PC=TPC \cup APC$ (plan components) is a system $G = (TPC, APC, P, Z)$ with

G_{PC} =	is the set of all doubly pointed graphs over *PC*;
TPC =	is the set of nonterminal symbols; (physical/virtual plan components)
APC =	is the set of terminal symbols; (atomic plan components)
P =	is a set of *CFERRs* over $PC = TPC \cup APC$ and G_{PC}
$Z\in G_{PC}$	is an initial graph; (which by application of *CFERR*s is successively decomposed creating the language of the *CFG*).

Since a plan component can often be expanded in different ways by applying different plan expansions, the system has to choose which one to take. For this purpose a plan expansion applicability function is attached to each plan expansion. The plan expansion applicability function (peaf) maps aspects of interest to a value [0...1] determining the degree of applicability. The process of choosing and applying an expansion rule is sketched Figure . A plan expansion is defined as:

Definition 9. Plan Expansion.A plan expansion pex_x*=(peaf, pef, app, CFERR)* is associated with a plan component class x such that it extends a *CFERR* by a plan expansion applicability function, a plan expansion function, and a plan expansion preconditon predicate.

peaf: parameters $\rightarrow$ *{True , False};* the plan expansion applicability function maps a parameter vector to True or False, depending whether or not this expansion is applicable for the given parameter vector.

pef: parameters $\rightarrow$ *new_parameters***;* the plan expansion functions determines the parameter vectors for all plan components within the right hand side of CFERR.

app: aspects_of_interest $\rightarrow$ *[0...1];* the applicability function app maps the operational context, such as the current situation given by sensor readings or the world model, to a heuristic notion of applicability of the plan expansion.

CFERR: a context-free edge replacement rule such that the left hand side of this rule equals x.

No matter how a decomposition of an abstract plan component is found it must be guaranteed that the decomposition does not conflict with already existing plan components in the current execution graph.

Thus, plan insertion implies not only the physical embedding of a subplan into the current execution graph but also the checking for possible conflicts.

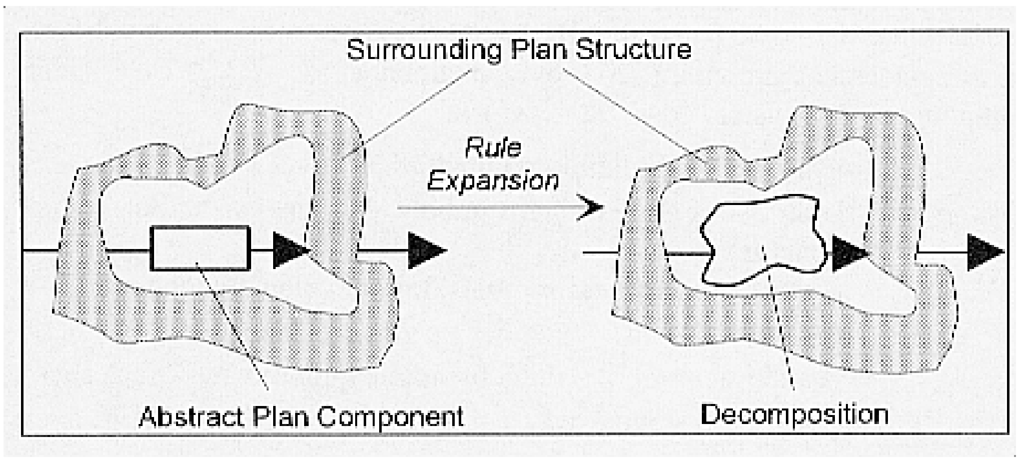


Figure27. Rule Expansion.

Embedding is analogous to context-free replacement of an edge (the abstract plan component) by a subplan. The subplan may have been selected from a set of production rules (as described above) or otherwise generated, e.g. by precondition achievement planning.

The application of a context-free edge replacement rule p yields a new current execution graph in which the plan component x with a plan component class pcc(x) which equals the left hand side of the edge replacement rule (lhs(p)) is replaced with the right hand side (rhs(p)) of the edge replacement rule. (Definition 10)

Definition 10. Application of Plan Expansions. Plan component decomposition is based on plan expansions. The plan expansion with the highest applicability value *app* is chosen.

The application of the context-free Edge Replacement Rule (*CFERR*) of this plan expansion maps a Source Graph *S* onto a Target Graph *T* ($S,T \in G_{PC}$) such that a plan component *p* within *S* with a plan component class matching the *lhs* of the *CFERR* is augmented by the right hand side *rhs* of the *CFERR* which is inserted between *from(p)* and *to(p)* in *T*.

Plan components new in *T* are instantiated (are given parameter vectors) by applying the plan expansion function pef with the parameter vector of *p*.

The process of embedding a subplan is completely local with respect to the surrounding plan structure (e.g. the current execution graph) as depicted in Figure.

4.8 EXECUTION AND FAULT RECOVERY

Execution control is based on life-cycle states and marked nodes in the current execution graph. Execution of abstract plan components involves decomposition, whereas execution of virtual plan components means monitoring.

4.8.1 Nominal Execution

Nominal execution denotes the case of error-free planning and execution. Figure shows a abstract plan component (state = "planned") which is decomposed during planning. The decomposition of the plan component is later executed. After all subordinate plan components have finished execution, the decomposition is removed and the abstract plan component's state is changed to "finished".

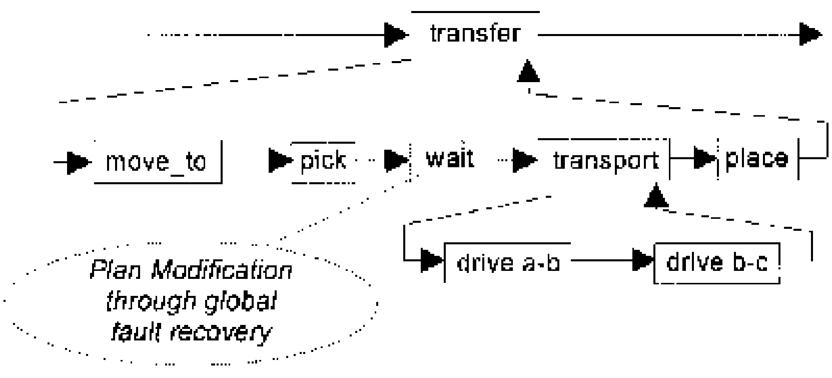


Figure 28. Nominal Execution.

4.8.2 Exception Execution

Exception execution implies that something did not go as planned during execution. Reasons are manifold:

- **Monitoring Alert** issued by a virtual plan component:

  I-Virtual Plan Component: Execution of the plan component stops.

  T-Virtual Plan Component (physical plan component failure): This situation indicates that the constraints under which safe and correct execution of the related physical plan component is assumed are violated. Consequently, the related physical plan component must immediately stop execution, although in many situations merely stopping is not the brightest thing to do.

  T-Virtual Plan Component (causal link destruction): Execution of the plan component stops.

G-Virtual Plan Component: Execution stops.

- **Execution Failure** of a physical plan component:

  Abstract Plan Component, i.e. failure to decompose a new plan:

  Atomic Plan Component, i.e. execution failure: Execution of the plan component is stopped.

After execution has been suspended error recovery takes place. We propose three ways to handle exceptional situations, that is fallback, local recovery, and global recovery:

- **Fallback:** The simplest way, though not really a solution, is to fallback to the next higher level of abstraction and replan. This is accomplished by unrolling the planning process and setting the next higher level's abstract plan component's state to "Error". In case unrolling involves abstract plan components, all plan components in their decomposition must terminate, shut down, or at least suspended. This forced termination is done recursively if necessary. Figure sketches this situation. After fallback, replanning is tried for the abstract plan component.

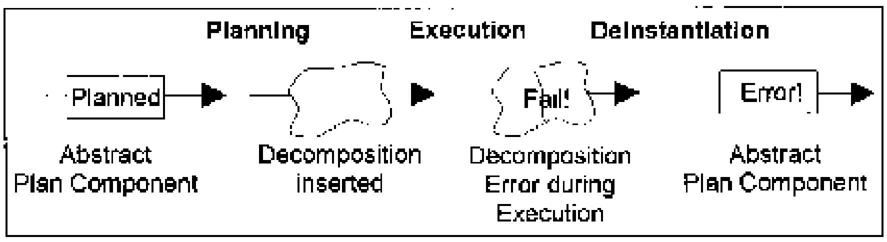


Figure 29. Fallback.

- **Local recovery:** In contrast to fallback, local recovery is a true fault recovery strategy. Recovery is carried out by providing recovery strategies which replace an unsuccessful decomposition by a hopefully better one. In order to carry out local recovery, recovery strategies must be readily available within the plan component class of the abstract plan component as a function which maps the failure pattern of the failing decomposition to a recovery strategy (local recovery plan). Information about the failure must be available to the recovery strategy. See Figure for illustration.

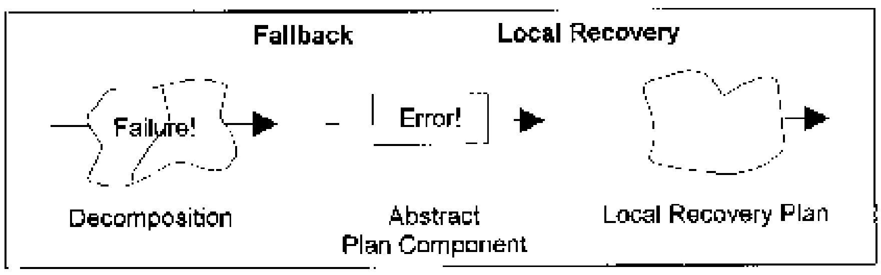


Figure 30. Local Recovery.

- **Global recovery:** Certain situations, such as low battery charge levels or an object entering the working area of the robot, are best addressed using global fault recovery schemes. Global fault recovery is useful in situation when the insertion of (abstract) physical plan components (a global recovery plan) into the current execution graph at the current point of execution resolves all problems so that the formerly planned activities can proceed unchanged as soon as the global recovery plan has finished execution. A low battery charge level indicated by a T-virtual plan component could then be easily addressed by inserting an abstract plan component "recharge batteries". After recharging the batteries execution can continue as planned earlier. (However there might be more efficient solutions involving route replanning from the recharge location.) Similarly, an exception caused by a moving object within the work area of the robot can then be addressed by inserting a plan component "inform operator and wait". Figure depicts this case. The insertion operation is described in detail in Section 4.9.

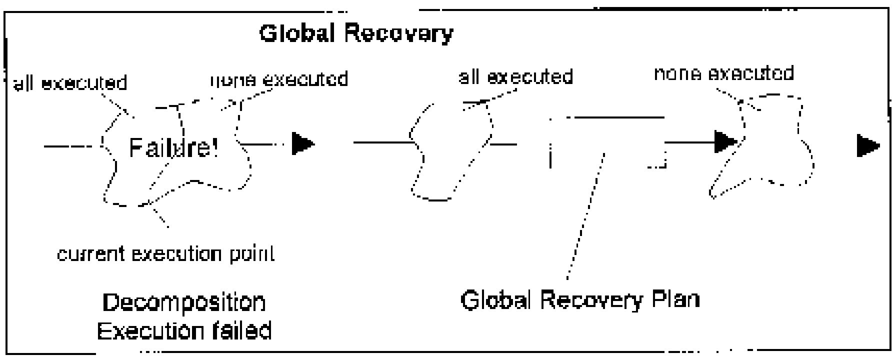


Figure 31. Global Recovery.

4.9 INTENTION SWITCHING

Intentions naturally change over time. New activities are planned, old activities are completed, or failures (detected by the monitoring system) cause the system to

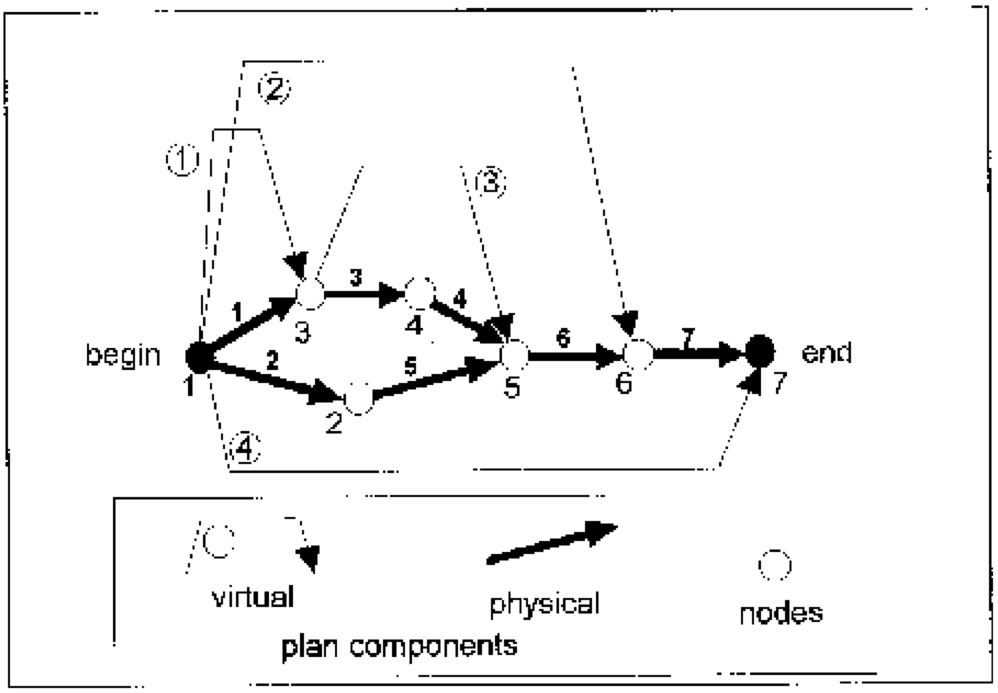


Figure 32. Determining Insertion Points.

fallback. Moreover, intention switching may cause the execution context to be switched.

What happens if a robot needs to do something more important than what is currently planned? Obviously, it should interrupt its current activity and start to carry out the more important job. This seems to be a desirable functionality of robot control systems. However, intention switching (which is synonymous with task and behaviour switching) has not previously been addressed as a functionality of robot control architectures.

The idea of intention switching is to insert the new intention at a suitable place. A suitable place is a place in the current execution graph that immediately precedes only "planned" plan components at which no conflicts with existing plan components exist.

A suitable place for insertion can be found by checking existing virtual plan components within the plan with those to be added. If a conflict might appear the plan component to added must be ordered either before or after the conflicting virtual plan component. This procedure leaves space for future improvement, as most of the time it is not known how an abstract plan component will be decomposed and what resulting virtual plan components are going to be added.

Figure shows an execution graph with 4 virtual plan components and 7 physical plan components. For inserting a new plan component, all virtual plan compo nents within the current execution graph which have not completed are checked for possible conflicts (as described in Plan Validity Monitoring II).

Since the point where the particular decomposition of the physical plan component to be inserted is not determined (or even possible to determine), assumptions and perhaps educated guesses about how the decomposition will proceed and what constraints will be involved must be used to estimate which particular insertion points are most suitable.

If the virtual plan component "3" is not compatible with the new plan component (i.e. there is a conflict between their conditions) then the new plan component must be executed (i.e. ordered within the current execution graph) before or after virtual plan component "3" (and of course its related physical plan components). If, additionally, virtual plan component "1" conflicts with the new plan component, further ordering constraints are imposed. With *n* denoting the new plan component and "1" and "3" the virtual plan components the constraints are:

- $n < 1$ and $n < 3$: Inserting the new plan component right in the beginning is only possible if the state of plan component "1" is "planned". (Figure)

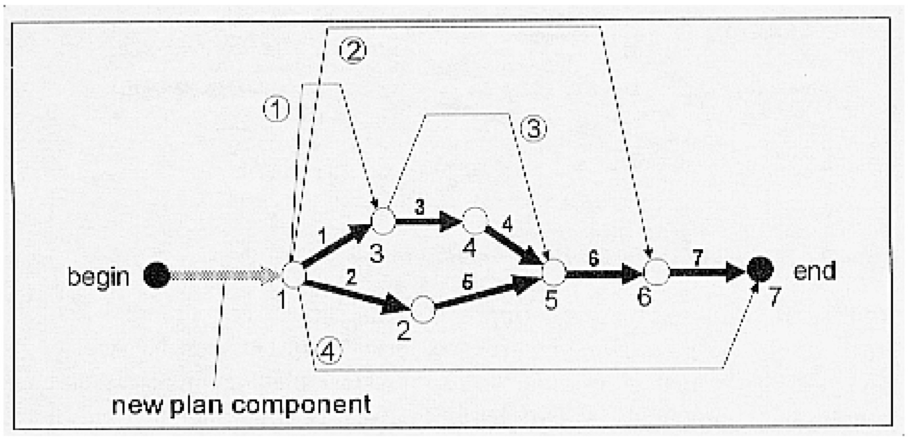


Figure33. Intention Switching (Insertion before node 1).

- $n < 1$ and $n > 3$: not possible because of $1 < 3$.

- $n > 1$ and $n < 3$: This causes the node "3" to be split. (Figure)

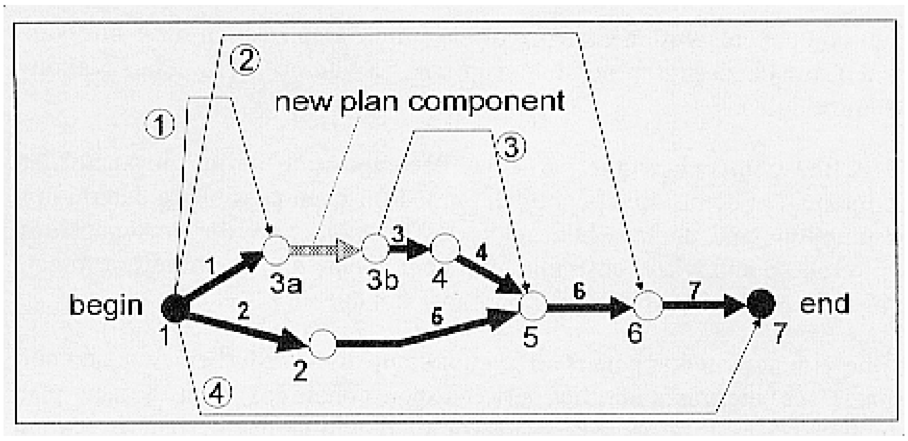


Figure 34. Intention Switching (Insertion between components 1 and 3).

- *n* > 1 and *n* > 3: This simply adds *n* between “5” and “end”. (Figure)

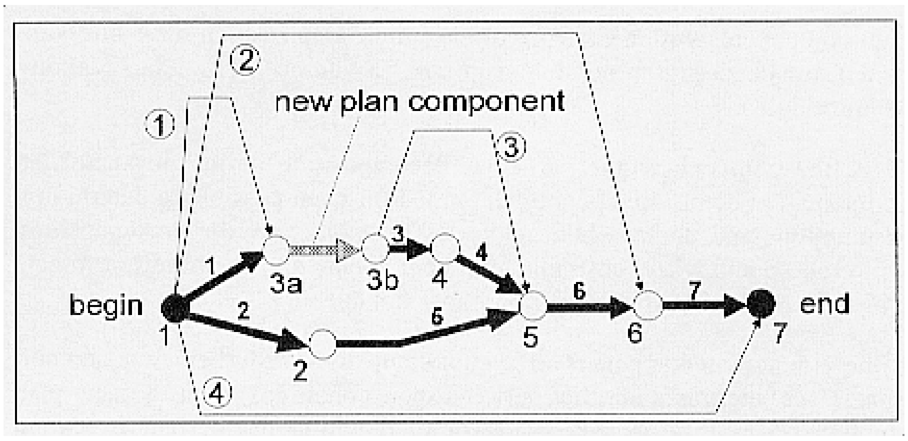


Figure 35. Intention Switching (Insertion between 5 and 7).

Additional orderings may be imposed if exclusive execution is desired. Furthermore, no plan component can be inserted in between I-virtual component and physical plan components, physical plan components and G-virtual plan components, or parallel to T-virtual plan components but in sequence with its related physical plan component.

After all ordering constraints have been determined, the best of all possible places of insertion must be selected. The criteria for determining the best insertion place may be several and a best compromise, e.g. between cost, risk, deadlines, priorities, etc. must be found. As with the applicability of production rules during decomposition of abstract plan component we propose to use a heuristic for this part of intention switching.

Sometimes, a new node must be created within the current execution graph due to insertion of a new plan component (intention) as in Figure and Figure . Since insertions can only take place at places, which execution has not reached, these nodes do not carry any marks.

Obviously, inserted intentions may cause problems during the deinstantiation process since the inserted intentions may arbitrarily split up the decomposition tree (since it might be inserted in between two plan components which belong to the same decomposition). In case of nominal execution there is no problem because deinstantiation takes place as plan components finish execution.

In cases of errors, fault recovery and deinstantiation are carried out locally with respect to the faulty intention. Intentions embedded at lower levels of representations are "shifted up", i.e. the decomposition is reduced to the abstract plan component, as depicted in Figure .

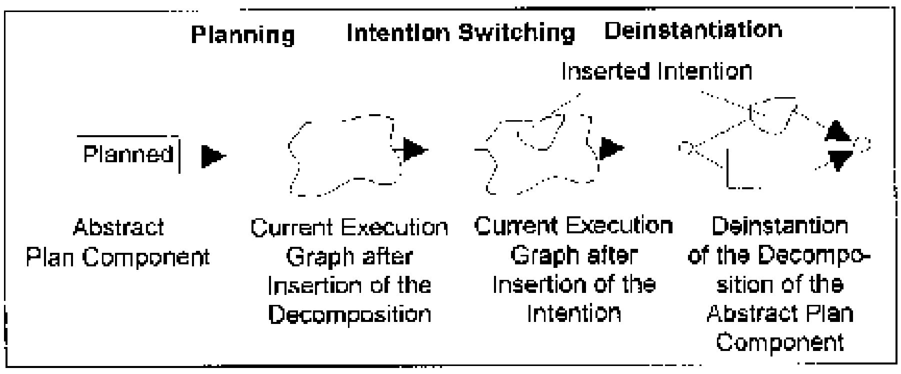

Planning
Intention Switching
Deinstantiation
Inserted Intention
Planned
Abstract Plan Component
Current Execution Graph after Insertion of the Decomposition
Current Execution Graph after Insertion of the Intention
Deinstantion of the Decomposition of the Abstract Plan Component

Figure 36. Deinstantiation.

If nodes have been split during plan component insertions (e.g. nodes 3a and 3b in Figure), these nodes are remerged when the inserted plan component is removed from the plan during fault recovery. Under normal circumstances the "before" parts of split nodes are executed and removed before execution of the inserted intention starts.

Sometimes intention switching results in problems because virtual plan components within the current execution graph conflict with those expected to be added due to the insertion of a new intention (abstract plan component). Some of these problems can be alleviated by simple plan adaptation:

- **Serialising**: Computational or other resources may run low if to many activities are started within a small interval. Serialising execution resolves some of the problems.

- **Information Goals**: Knowledge about certain conditions may not exist. Inserting a plan component for acquiring the information resolves this problem.

- **Shutdown and Restart**: Sometimes protection intervals indicate the use of a resource which could be temporarily freed and used for some other purpose provided its status is restored after use, e.g. a robotic arm holding an object could be used for some other purpose if the object is temporarily placed somewhere (shutdown) and re-grasped afterwards (restart).

Both plan adaptation through information goals and shutdown and restart can be conceptually modelled by plan component insertion(s).

4.10 MONITORING

Planning and execution use a plan representation which represents the dynamic world as a discrete event system. Within this representation continuous events are subject of monitoring only within the scope of T-virtual plan components. The approach taken within this thesis identifies two dimensions of monitoring: *time* and *concurrency*. The key concepts of monitoring are:

- *Plan execution monitoring* and *plan validity monitoring* both concerns monitoring of plan execution within the space spanned by time and concurrency.

  - *Plan execution monitoring* is the task of detecting conflicts and abnormal behaviour during execution. This task ensures the correctness and safety of execution of single plan components, and consequently the basic operability of the robot. To accomplish this subtask, plan component behaviour information is matched against sensor data using the monitoring predicates defined in Chapter 4.5. This plan component behaviour information must be as complete as possible and should include pre- and post-condition information about transient states. (See Chapter 4.10.1.)

  - *Plan validity monitoring* is the task of checking that the actual plan is a valid plan with respect to the actual world state. Plan validity again is a twofold task: (See Chapter 4.10.2.)

    - to ensure that plan insertion is correct. Hence, plan validity monitoring is called on plan component insertions.

    - to ensure that assumptions on which plan validity is based upon, i.e. precondition - postcondition relationships, are valid. For this purpose, established subgoals and preconditions must be continuously checked for their validity based on sensor readings and plan component behaviour information again using the monitoring predicates defined in Chapter 4.5.

- Monitoring two different ontologies of dynamic systems: *discrete event systems*, described by plans, and *continuous event systems*, described by plan components, involves monitoring discrete state information such as the existence of certain (pre- or post-) conditions, as well as continuous state transitions, such as qualitatively and/or quantitatively monitoring a behaviour. As illustrated in Figure monitoring includes both discrete states and continuous behaviours.

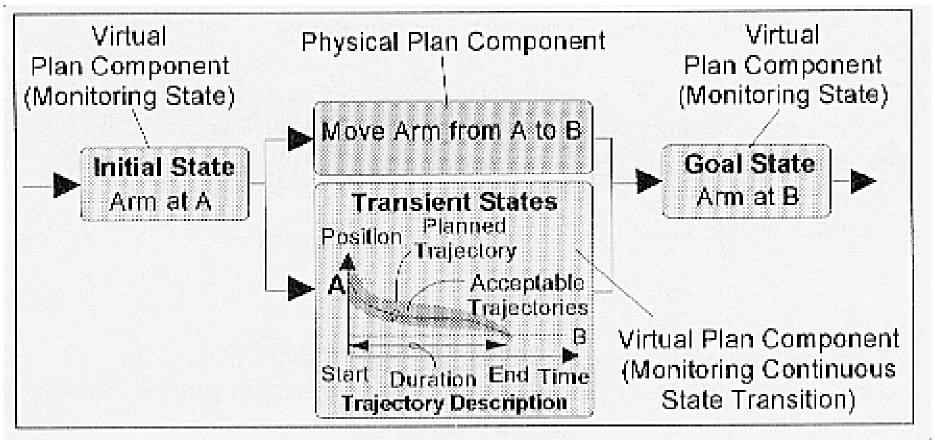


Figure 37. Monitoring Plan Execution and Plan Validity.

4.10.1 Plan Execution Monitoring

Plan execution monitoring takes place during the execution of a plan component, including its starting and finishing phases. Its task is best described as continuously comparing the current world and system state with the predicted and planned behaviour described by virtual plan components. Plan execution monitoring denotes the surveillance of plan execution during the three phases of execution: start up, execution, and completion.

For each phase different monitoring activities must be carried out and different model data must be used. These pieces of information are provided by the descriptions as the *initial*, *transient*, and *goal state* descriptions held by the I, T, and G virtual plan components.

The monitoring activities are provided as functions implementing the monitoring predicates given in Chapter 4.5 within the plan component class definitions. Simply executing them ensures the validation of the model information expressed in the domain dependent description language.

Summarising, the different monitoring activities for plan execution monitoring are:

- **I-Virtual Plan Components:** Validation of propositions (x) listed as a pre-condition for a succeeding physical plan component. This proposition must match with sensor readings (s). These checks take place before execution of the related physical plan component. It should be made sure that the delay between the execution of the I virtual plan component and the succeeding physical plan component is as short as possible.
  A problem is indicated if

  $(\exists x \in \mathit{initial\_state}$ such that $\mathit{not\_matching}(x,s))$

- **T-Virtual Plan Components:** During execution of a physical plan component the transient states are validated by executing T virtual plan components at regular intervals until the physical plan component finishes execution.
  A problem is indicated if

  $(\exists x \in \mathit{transient\_state}$ such that $\mathit{not\_matching}(x,s))$

- **G-Virtual Plan Components:** After execution of a physical plan component has finished, the achieved system state has to be verified by executing the succeeding G-virtual plan components.
  A problem is indicated if

  $(\exists x \in \mathit{goal\_state}$ such that $\mathit{not\_matching}(x,s))$

4.10.2 Plan Validity Monitoring

Plan validity monitoring is the task of determining whether a given plan is correct with respect to a changing world. It checks whether beliefs about the world based on sensor readings match the internal world model.

Plan validity monitoring is based on checking postcondition-precondition relationships (causal links) and possible interactions of initial, transient, and goal state descriptions of parallel plan components in non-linear plans. If postcondition-precondition relationships are represented within the plan structure as virtual plan components (as proposed earlier) plan validity monitoring reduces to checking the virtual plan components.

If these relationships are not represented within the plan structure they can be determined as follows:

- Checking insertion of plan components and subplans for correctness.
- Verifying assumptions (postcondition - precondition relationships) underlying the plan structure.

Checking plan insertion is described earlier. Assumptions underlying plan validity are represented by T-virtual plan component within the current execution graph. Their regular execution, at least once, ensures the detection of corrupted plans which may fail in future due to incorrect assumptions about the environment.

4.11 JANUS - FINDING CAUSAL DEPENDENCIES

Causal dependencies are not always readily available. Here, we propose a method to derive dependencies from operator descriptions and to use them for monitoring. In contrast to the representations described above this method does not use virtual plan components but entirely relies on operator descriptions with explicit initial, transient, and goal state descriptions. This representation is nevertheless equivalent with the previous one.

Plan correctness for non-linear plans (not involving possible unforeseen changes in the world) is determined by the Modal Truth Criterion (MTC; Chapman 1987). The notation of the MTC involves the notation of *establishers* and *violators* (See Chapter 2.1.1.5) Violators of a plan component *x* are plan components which are carried out before *x* and which undermine at least one prerequisite of *x*. Establishers of a plan component *x* are plan components which may be carried out before *x* and which establish a prerequisite of *x*. See Figure . Their mathematical definition is given in Definition 11.

Definition 11. Establisher and Violators. A plan is given as a set of nodes N and a set of precedence relationships *R*. A precedence relationship $(p, q) \in R$ denotes that *p* occurs prior to *q* within the plan, $p,q \in N$. *k* and *l* are propositions. $goal\_state_q$ is a description of the state changes after execution of *q*. Although this definition of plans is different than the one given earlier for plans as doubly pointed graphs, both representations are equivalent and can be mutually translated into each other without loss of generality with respect to the following definitions.

Establishers are then defined as:

$$q \in Establishers(p,k) \Leftrightarrow consistent(k,l) \wedge l \in goal\_state_q \wedge (p, q) \notin R.$$

Violators are defined as:

$$q \in Violators(p,k) \Leftrightarrow not\_consistent(k,l) \wedge l \in goal\_state_q \wedge (p, q) \notin R.$$

In order to check the MTC, plan validity monitoring must be carried out for each proposition which is part of the initial state description for each plan component. This task is split into two pieces. First the sets of violators and establishers are determined. In the second step, plan validity monitoring is carried out as described below.

"Plan Validity Monitoring (I)" essentially employs the MTC in order to check plan validity. (See Definition 12)

Definition 12. Plan Validity Monitoring (I). A plan is given as a set of nodes N and a set of precedence relationships R. A precedence relationship $(p, q) \in R$ denotes that p occurs prior to q within the plan, p,q $\in N$.

Possible conflicts are anticipated, if one of the following conditions hold true:

$\exists k\ \forall x \in Establishers(p,k)\ \exists y \in Violators(p,k) \Rightarrow (y,x) \notin R$, i.e.,

> for all establishers there exists at least one proposition k of a plan component p, such that there exists at least one violator y of k, which is possibly executed after x .

$\exists k\ Establishers(p,k) = \emptyset \wedge Violators(p,k) \neq \emptyset$, i.e.,

> at least one proposition k of a plan component p exists which has no establisher but at least one violator.

Pending conflicts are anticipated, if

$\exists k\ \exists y \in Violators(p,k)\ \forall x \in Establishers(p,k) \Rightarrow (x, y) \in R \wedge (y, p) \in R,$

> i.e., if for at least one violator y of at least one proposition k of a plan component $p,$ all establishers are planned to be executed before the violator. This condition indicates a planning error and can be removed if the planner produces only correct plans, e.g. TWEAK or ABTWEAK. However, if this is not guaranteed it is worthwhile to check this condition, too. The case that the violated condition may be reestablished by an unknown outside effect is ignored here.

$\exists k\ Establishers(p,k) \neq \emptyset \wedge (\neg matching(k,s) \vee \exists x \in Violators(p,k) \Rightarrow (x, p) \in R),$

> i.e., if for at least one proposition k of a plan component p no establisher exist and the proposition k does not match with the or sensor readings or at least one violator exists which is executed prior to p.

During "Plan Validity Monitoring (II)" relationships among concurrent plan components are checked. However, two plan components which may be executed concurrently may also be executed sequentially. Through proper serialisation of parallel plan components some of the problems indicated by Plan Validity Monitoring (II) can be resolved. (See Definition 13)

Definition 13. Plan Validity Monitoring (II). Plan components which may be executed concurrently have to be further checked for conflicts.

A plan is given as a set of nodes *N* and a set of precedence relationships *R*. A precedence relationship $(p, q) \in R$ denotes that *p* occurs prior to *q* within the plan, $p, q \in N$.

A conflict is imminent if at least one pair of (parallel) propositions exists which is not consistent ($p, q \in P$, $(p, q) \notin R$, $(q, p) \notin R$, $status_p$ = *"planning"* , $status_q$ = *"planning"*):

$\exists k \in m\_conds_p \wedge pc\_type_p \in$ {T-virtual, I-virtual, G-virtual} $\wedge$
$\exists l \in m\_conds_q \wedge pc\_type_q \in$ {T-virtual, I-virtual, G-virtual}
$\Rightarrow not\_consistent(k, l)$, with $pc\_type_q = pc\_type$ of class of *q*.

The theoretical aspects of monitoring, particularly the determination of causal links and plan validity monitoring, have been used to design and develop a pilot implementation of a monitoring shell, called JANUS (Hasemann 92). This monitoring shell (which in analogy to expert system shells lacks application-dependent knowledge, e.g. no domain-specific description language is defined) has been tested in a simulated environment. The results have exhibited the desired behaviour and the ability to forecast conflicts prior to execution.

4.11.1 Situation Grids

The central data structure used inside JANUS is the Situation Grid. The Situation Grid is used to track the ongoing planning and execution process. Incoming planning and execution messages are processed and entries in the Situation Grid are made.

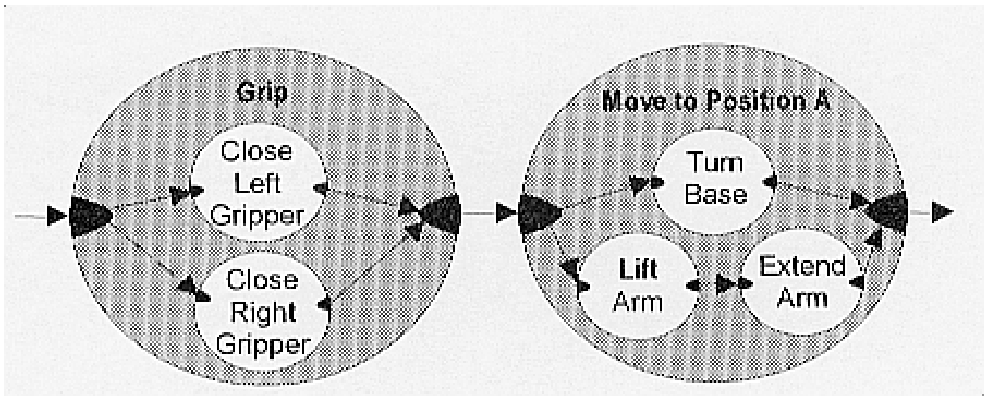


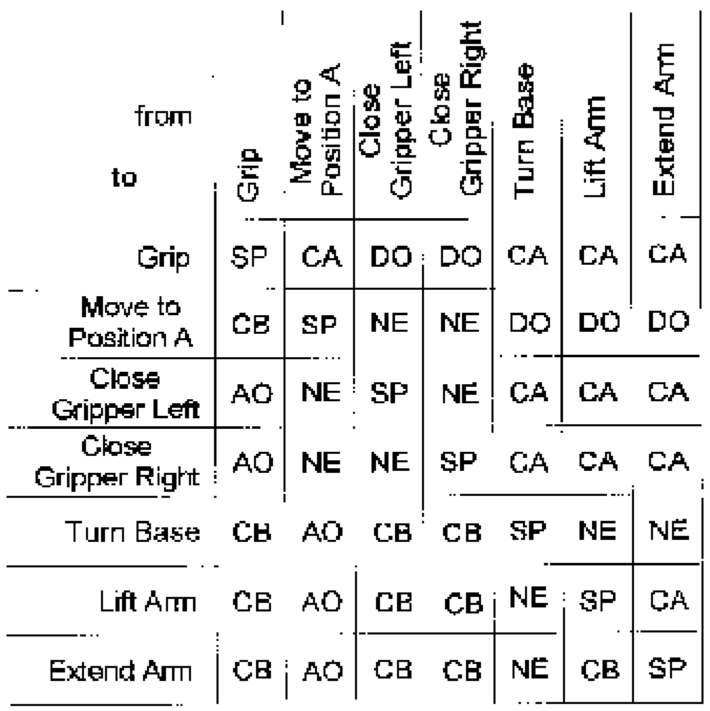
Figure 38. A Plan and its Relationship Matrix.

The Situation Grid is the only place of communication and data exchange between servers and clients. The server maintains the Situation Grid and Clients receive information about what to monitor on the basis of information stored in the Situation Grid. The Situation Grid is an array which can grow and shrink dynamically, however plan components must be created and deleted in chronological order.

For each plan component scheduled or currently being executed, one column and one row inside the Situation Grid exists. The cells of the Situation Grid reflect the sequentiality relations among plan components. Conceptually, there are six different relations possible:

- **No Entry (NE):** Two plan components are in a NE-relationship, if and only if they are allowed to be executed concurrently.
- **Ancestor of (AO):** Two plan components are in a AO-relationship, if and only if the first one is an ancestor of the second one.
- **Descendent of (DO):** Two plan components are in a DO-relationship, if and only if the second one is a ancestor of the first one.
- **Comes before (CB):** Sequentiality constraints determine the sequence in which plan components must be carried out. Thus, two plan components are in a CB-relationship, if and only if the first one must be carried out before the second one.
- **Comes after (CA):** Two plan components are in a CA-relationship, if and only if the first one must be carried out after the second one.
- **Same PID (SP):** SP denotes the plan component's relationship with itself. This relationship is not used.

We denote a relationship between two plan components “A” and “B” with (A,B)=XX with XX being one of the above described relations. Figure depicts a simple plan and Figure its relationship matrix.



from / to	Grip	Move to Position A	Close Gripper Left	Close Gripper Right	Turn Base	Lift Arm	Extend Arm
Grip	SP	CA	DO	DO	CA	CA	CA
Move to Position A	CB	SP	NE	NE	DO	DO	DO
Close Gripper Left	AO	NE	SP	NE	CA	CA	CA
Close Gripper Right	AO	NE	NE	SP	CA	CA	CA
Turn Base	CB	AO	CB	CB	SP	NE	NE
Lift Arm	CB	AO	CB	CB	NE	SP	CA
Extend Arm	CB	AO	CB	CB	NE	CB	SP

Figure 39. The Relationship Matrix of the Plan depicted in Figure .

These relationships between two plan components are determined whenever a new plan component is added. This evaluation is based on the sequentiality constraints attached to each newly added plan component, the decomposition path, and the plan components already stored in the situation grid.

4.11.2 Inheritance of Relationships

Whenever a new plan component is planned in JANUS (planning is restricted to adding plan components to the end of the plan and to refining plan components one plan component at a time), a message is sent to the Situation Grid which contains the plan component's description, the list of its direct predecessors, and its parent.

Using this information the plan component's relation to other plan components can be determined by the rule base shown in Table 7. The rules are applied to determine the relations of the newly added plan component to other plan components already stored in the Situation Grid. Table 6 describes the symbols used in Table 7.

Table 6. Symbols used in Table 7.

Symbols used	
A	Plan component:
CC(A)	Causality constraints:
PA(A)	Parent Plan Component ID:
PID(A)	Plan Component ID:
Y	Plan Component to be added:
X	Other Plan Components:
(X1, X2)	Relationship Matrix element:

Table 7. Rules for Updating the Relationshipmatrix.

Rules:	
1.	$(Y,Y) = SP$
2.	$X \in CC(Y) \Rightarrow (X,Y) = CB, (Y,X) = CA$
3.	$(X, PA(Y)) == AO \Rightarrow (X,Y) = AO, (Y,X) = DO$
4.	$(X, PA(Y)) == DO$ and $X \notin CC(Y) \Rightarrow$
	$\exists x \in CC(Y)$ and $(x, X) == AO \Rightarrow$
	$(X,Y) = CB, (Y,X) = CA \vee (X,Y) = NE, (Y,X) = NE$
5.	$(X, PA(Y)) == CA \Rightarrow (X,Y) = CA, (Y,X) = CB$
6.	$(X, PA(Y)) == CB \Rightarrow (X,Y) = CB, (Y,X) = CA$
else	$(X,Y) = NE, (Y,X) = NE.$

Plan monitoring can become very time consuming for plans with a large number of plan components. Within JANUS (Hasemann 1992) we implemented an efficient method to distribute and parallelise monitoring among World State Aspect Objects. Each World State Aspect Object carries out monitoring for its own domain. Assuming that conflicts always appear within the domain of at least one World State Aspect Object, correct monitoring can be ensured.

4.11.3 Results from a Pilot Implementation

JANUS has been successfully tested using different simulated planning and execution sequences. In these test runs, simple plans for the blocks world pick and place domain were described using STRIPS-like operators. The test runs consisted of simulated planning and execution sequences, simulated sensor readings and the protocols which give information about the time each message has been sent or received.

During the creation of plans and later execution a number of errors were simulated, such as:

- **conflicting parallel activities:** This type of error exists if execution of parallel plan components in a certain order or concurrently would result in a conflict. JANUS detected conflicting parallel plan components during the creation of plans and notified the control system about critical execution sequences.

- **invalidated preconditions** hinder the execution of related plan components. JANUS checked preconditions before execution of activities and detected missing preconditions.

- **invalidated postconditions** result from unsuccessful execution of plan components. JANUS checked postconditions after execution of activities and detected these failures promptly.

- **invalidated transient states:** These results are failures which occur during execution of activities. JANUS checks transient states during execution at regular intervals and detected the problems.

- **invalidated causal links:** This error results from the deletion of established subgoals during plan execution. During the test runs, deletion of subgoals was simulated and detected by JANUS shortly thereafter.

Although the task of monitoring within these simulations was rather easy, it was complicated enough to illustrate the capabilities of JANUS's monitoring subsystems. A more complete presentation of the test runs, their protocols, and results can be found in (Hasemann 1992).

4.11.4 Comments on JANUS

JANUS uses situation grids to keep track of the plan creation and plan execution processes. However, due to the use of situation grids, plan components must be created and deleted in chronological order. In the proposed robot control architecture this can not be assumed.

For this reason JANUS could not be integrated within the implementation of our robot control architecture. Nevertheless, the theory of plan validity monitoring developed for JANUS remains valid.

One lesson learned from JANUS was that the value of automated determination of postcondition - precondition relationships (causal links) is questionable for two reasons: (a) possible ambiguities and interaction among causal links and (b) it seems to be a return to the fruitless avenues of classical AI.

Reason (b) in particular, is a point of criticism, since most of the time knowledge about plan execution constraints exists at design time, should make its way into the representation of plans and operators, and for this reason make automated determination of postcondition - precondition relationships unnecessary.

The proposed robot control architecture avoids this caveat of AI operator descriptions by allowing the designer to specify and protect causal links by virtual plan components within the subplan (decomposition) descriptions for abstract plan components.

Nevertheless, the methods for deriving causal links from plan structures are not completely useless and can provide valuable information for a system designer at *design* time by analysing designed plan templates and suggesting causal links.

4.12 OPERATING SYSTEM INTEGRATION

Embedding the proposed architecture into an operating system is straightforward. The following processes are identified (Figure):

- **Execute Plan Component**: Execution of virtual plan component implies monitoring, execution of atomic plan components corresponds to execution at the driver level, and execution of abstract plan components means decomposition (planning). Execution processes are created by “Clean Up”.

- **Check for Errors**: This process checks the current execution graph for errors. If errors are found the error recovery of the concerned plan component is invoked.

- **Check Trigger Conditions**: This process continuously checks the trigger conditions for all behaviours (plan component classes with trigger conditions). In case a behaviour is triggered, its insertion into the current execution graph is attempted.

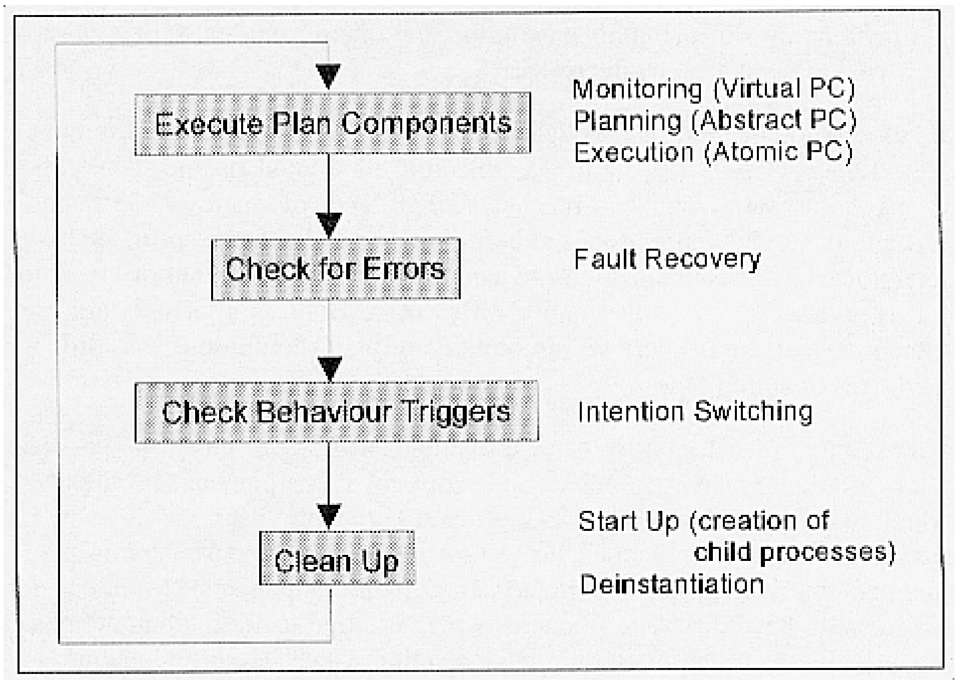


Figure 40. Main Processes.

- **Clean Up**: This process checks the current execution graph for plan components ready to be executed, generates a new child processes for any execution process, and schedules them. This process also checks for successfully executed plan components and removes them (if possible) from the current execution graph.

Cooperative scheduling, i.e. each process runs until it returns control, can be applied if the execution time for each call is guaranteed to be noncritical. Although far more easy to implement, cooperative scheduling is dangerous in situation in which processes may take longer than expected or may not return control at all. Preemptive scheduling in which a central scheduler allocates time slices to each process and schedules them, offers an alternative.

Write access to the current execution graph must be mutual exclusive, i.e. during modification of the current execution graph (plan component insertion, fault recovery) only one process is allowed to update it. Since fault recovery and plan component insertion may take significant time it must be assured that they do not occur during critical intervals (e.g. during the execution of atomic plan components with time critical control loops) but are postponed until time for fault recovery can be allocated (e.g. until the atomic plan component has finished execution).

4.13 DISCUSSION

We have proposed a robot control architecture which covers the indispensable functionalities of planning, execution, and monitoring, as well as intention switching and fault recovery.

The closest intellectual relative is the Task Control Architecture (Simmons 1990, 1994). TCA is the control architecture of a good dozen of autonomous robot systems and has reached a high-level of maturity. Although TCA and the architecture proposed here share much the same spirit, such as hierarchical task decomposition, sequencing of subtasks, monitoring, and fault recovery, there are also major differences. Both architectures are restricted to (loop free) linear representations of plans composed from different classes of action types.

Within the proposed architecture we distinguish between physical, i.e. abstract, or atomic, plan components and virtual plan components (I,T, and G-virtual plan components). TCA, on the other hand, uses Goals, which roughly correspond to abstract plan components, Commands (atomic plan components), and explicit monitors (virtual plan components). Monitors in TCA usually have different outcomes which are tied to succeeding actions. This introduces conditionality in plan execution which in our architecture is performed by selecting the most suitable subplan when decomposing abstract plan components. TCA distinguishes further between single-shot (I and G virtual plan components), and periodic monitors (T-virtual plan components).

Intention switching does not exist as such in TCA, though there are several techniques for introducing reactivity by means of exceptions. Exceptions are raised by monitors and are handled by exception handlers which are attached to monitoris. The search for handlers is hierarchical, starting from the exception-issuing plan component upwards in the task tree until a handler is found. Reaction to exception usually results in a modification of the task tree as within our proposed architecture.

Another conceptual difference are the delay planning constraints within TCA. Delay planning constraints define when execution of planning tasks may occur. In our proposed architecture this could be done using virtual plan components.

Furthermore and most important, monitors in TCA are used for checking conditions only, while in our architecture virtual plan components additionally play an important role by describing the constraints for plan manipulation during plan component insertion and fault recovery.

The proposed architecture does not handle repetitive actions or action sequences explicitly. This may or may not cause problems with particular applications. Introducing loop structures would significantly increase the complexity of plan insertion and thus has been avoided as we believe the disadvantages of introducing loop structures far outweigh their conceptual benefits. Workarounds to this problem exist. Repetitive tasks can be modelled in several ways:

- using the trigger function to enable a repetitive activity as long as it is necessary,
- by “abusing” fault recovery, i.e. the repetitve activity results in an “erroneous condition” which is tied to a recovery routing which repeats the activity as long as necessary.
- for small number of repetions, plan templates with multiple instantions of the repetive activity could be created.
- as a last resort, repetions could be modelled by recursive decomposition of abstract plan components, though this is not recommended for high numbers of repetitions as the current execution graph may become very big.

It is matter of further study, if and how end recursions in plan component decompositions could be resolved and how this affects fault recovery.

5 EXPERIMENTS AND RESULTS

The robot control architecture has been implemented and tested in a simulated harbour environment. See Figure for illustration. Although the use of simulations may be questionable in the domain of an empirical science such as robotic research, we claim that it proved useful for evaluating the dynamics of planning, execution, and fault recovery mechanisms. The developed simulation environment allowed us to conduct numerous experiments at very low cost and in very short time compared to what would be involved in real world experiments involving a terminal area with one AGV and one paper roll manipulator. With respect to the symbolic levels of robot control such as task initiation, decomposition, and fault recovery, the architecture's response can be studied under very different conditions far faster than using real-time simulations.

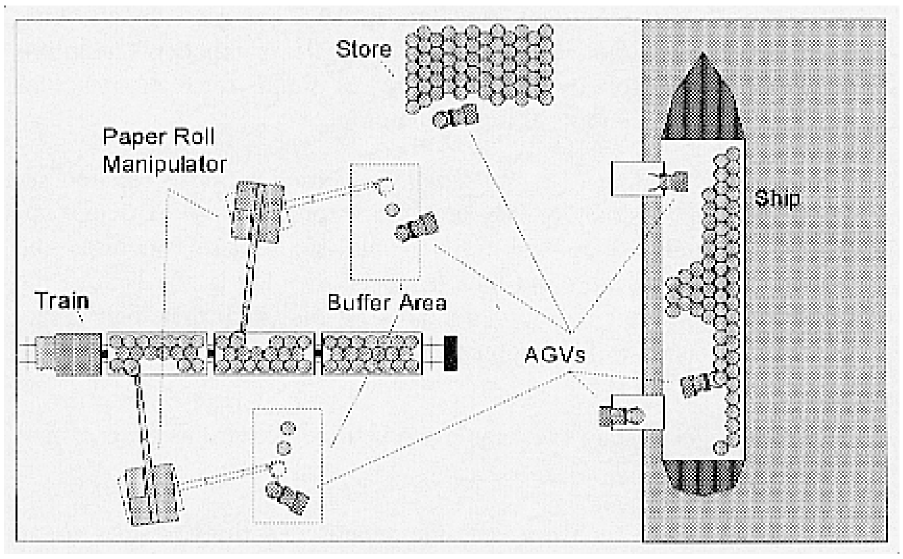


Figure41. Application Example.

The most compelling argument for using simulations is however the reproducability of results which is difficult, if possible at all, to obtain from real world experiments. Naturally, simulators exhibit serious disadvantages, particularly for modelling low-level activities and environmental responses. Care must be taken not to abstract away much of the complexity, dynamics, and uncertainties of the real world.

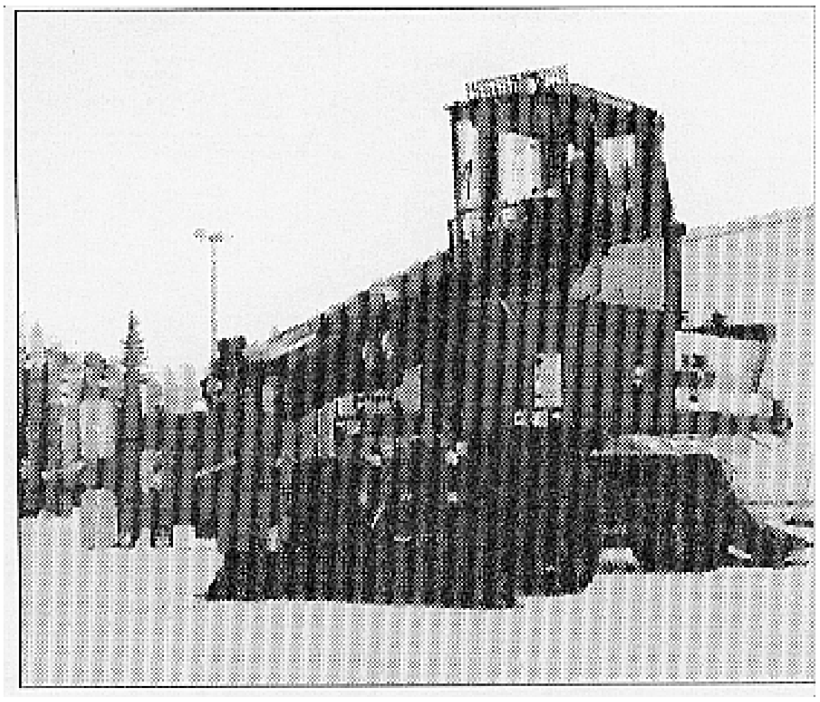


Figure 42. The Paper Roll Manipulator used in the Grasp and Motion Planning Experiments.

Aside from simultions covering the higher level activities, some of the lower level activities, in particular the grasp and path planning, geometrical reasoning, and fuzzy logic interpreter, have been integrated within the control system of the industrial paper roll manipulator shown in Figure .

5.1 ROBOT CONTROL ARCHITECTURE IMPLEMENTATION

The proposed robot control architecture was implemented and tested within a simulated environment consisting of a paper roll manipulator, a autonomously guided vehicle, a train car holding paper rolls to be unloaded, a truck to be loaded with paper rolls, and a fueling station for refueling.

Figure shows a screenimage of the simulation environment which consist of a World View displaying the simulated world, a Graph View which shows the current execution graph as well as the status of all currently planned plan component. A Monitor window displays the activation of each fuzzy rule during path and grasp planning and helps detecting flaws in the rule base during the design stage.

The objectives of this test environment are to try and apply the concepts of the proposed robot control architecture within an environment which comes close to envisaged applications of intelligent robots for service tasks regarding the nature and selection of tasks and events.

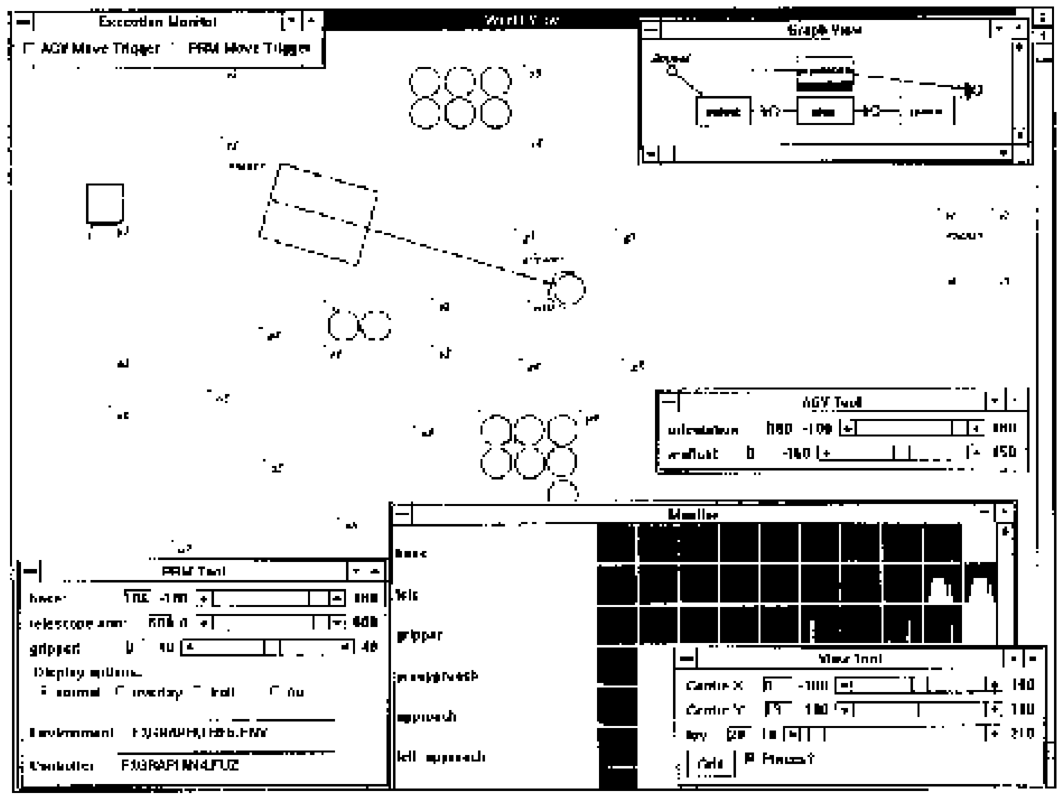


Figure 43. Simulation Environment.

This material flow scenario has been selected because the application of robots to load and unload trains and trucks is a real one, easily finds its correspondence in the real world, and exhibits many interesting features and properties to demonstrate the capabilities of the proposed robot control architecture. The features demonstrated include:

- **High-Level Tasks and Decomposition** - transporting paper rolls by picking, moving, and placing. Each of these subtasks are context dependent further decomposed into low-level tasks modelled by atomic plan components.

- **Low-Level Tasks and Control** - path and grasp planning and execution for the paper roll manipulator and the autonomous guided vehicle. In order to demonstrate the integration of lower level reactive components, we integrated the motion planning tasks of the paper roll manipulator and described them in detail.

- **Event Driven Activities** - insertion of triggered activities. Refueling is triggered when the fuel level is low. A suitable insertion point is sought and a refueling activity is inserted which in turn is further decomposed and executed.

- **Failure and Recovery**, the system handles simulated faults, such as blocked paths, and occasional conflicts by trying to insert conflict resolving activities or by falling back to the next higher level.

- **Operator Intervention** plays a primary role in recovery. Operator intervention is included as special plan components, e.g. when gripping a paper roll fails, or when decision support by the operator is needed, e.g. deciding whether the system should wait or plan an alternate route in case when the selected route is blocked. Operator intervention very naturally integrates into the proposed architecture.

The robot control architecture is embedded within a simulated environment. Both are implemented in C/C++. The simulator is used for visualization, world modelling, and providing a user interface to the system. It sits on top of the multi-platform GUI library wxWindows[1] under MSWindows 3.1.

The robot control architecture roughly consists of modules for execution control, graph manipulation for insertion and deletion of plan components within the current execution graph, and a set of base classes implementing default functionalities for the different plan component types.

Low-level behaviours and decision making is partly carried out using fuzzy logic. A fuzzy logic interpreter has been written to read and interpret fuzzy rule bases. Fuzzy logic rule bases and an environment description are stored in textual form with a lisp-like syntax. A parser conveniently interprets such textfiles. The fuzzy logic interpreter and the parser to read the fuzzy logic rule bases are written in C and are used in both the robot control architecture implementation and the control system of an industrial paper roll manipulator.

The fuzzy logic rulebases used for path and grasp planning are the same for the simulated environment and the real paper roll manipulator. In fact, the simulator used for the robot control architecture was at the same time used to develop, test, and fine tune the fuzzy logic rule bases used for path and grasp planning.

Although implementing a robot control architecture primarily served the purpose of empirically demonstrating its practical usefulness, it also provided additional insights on important implementation issues. One of the biggest assets of the architecture are perhaps the huge benefits that can be drawn from its object-oriented view of plan components. The taxonomy of plan components can be easily mapped into a class hierarchy in which subclasses, e.g. instantiations of atomic plan components, inherit basic functionalities from parent classes, e.g. from an abstract plan component class.

[1] this free C++ class library for platform independent development of graphical user interfaces, orignally developed at the Artificial Intelligence Application Institute, University of Edinburgh, is currently available via ftp from ftp.aiai.ed.ac.uk.

Application-related plan components inherit basic functionalities, such as insertion, from base classes and override virtual functions, such as execute, decompose, or fault recovery with application related code. By doing this, a tool box of functionalities is created which can be easily reused for other applications or environments.

In order to simplify the implementation and assist the integration of the simulated environment with the graphical user interface, scheduling of concurrent tasks is carried out cooperatively. Each process is executed and is not interrupted until it returns control to its parent process. Although this allows the evaluation system to a great extent to keep free from operating system details, a robot control architecture for a real application most probably must be built on top of a multithreaded operating system with a sophisticated scheduling in order to guarantee real-time constraints, at least for servo loops or environment surveillance tasks. However, real-time aspects are not addressed within the implementation.

5.2 FUZZY LOGIC FOR PATH AND GRASP PLANNING

The concepts of our proposed architecture address tasks of coordinating different activities: monitoring, execution, detection of abnormal behaviours, fault recovery, and planning. The concepts do not enforce any particularly technique for accomplishing these tasks but leave the designer freedom of how to implement decision making, planning, and control tasks.

A fast implementation was desired for the evaluation environment. We decided to implement major parts of the architecture using fuzzy logic primarily because of the character of fuzzy logic being able of bridging the gap between discrete event systems and continuous control as well as its power as a descriptive tool for describing complex behaviours as demonstrated by the influencing work of Saffioti et al. (1993a) in the field of mobile robot navigation.

We also had earlier good experiences with fuzzy logic in an project implementing an anti-slip system for off-road vehicles (Känsälä & Hasemann 1994a,b, 1995), (Hasemann & Känsälä 1994a,b), (Hasemann et al. 1994) and where familiar with the technology.

Thus we deemed fuzzy logic to be suitable for decision making: as trigger functions for determining the criticalities of different behaviours, as applicability functions for choosing the most suitable decomposition, and most important of all as tool for implementing complex skills such as grasp and path planning for the industrial paper roll manipulator shown in Figure . Grasping paper rolls is a challenging task, because of several objectives, such as:

- **Obstacle avoidance**. During the whole process of approaching and grasping a paper roll, collision with other paper rolls and obstacles must be avoided. See Figure .

- **Coordinated base, arm, and wrist motions**. Approach movements and particularly the grasping demand coordinated control of several joints.

- **Intelligent grasp**. Sometimes the gripper cannot grasp a paper roll directly but must move to some extent around the paper roll to avoid contact with adjacent paper rolls when moving its jaws around the selected paper roll. See Figure .

Although this task of motion planning could have been addressed by other methods, this method proved to be very easy to implement and adapt to different conditions, once the fuzzy logic interpreter was written.

5.2.1 Inputs and Outputs

The grasp planning behaviour is described by a set of fuzzy rules which are stored in a text file and are loaded by the control architecture on program start. Consequently, the planning behaviour can be easily changed by editing the rule bases and loading them into a simulation environment. In the simulation, planning scenarios can be created, modified, loaded, and retrieved. Figure depicts the control parameters of the fuzzy logic controller for grasping paper rolls, which are:

- $\Delta\alpha$, the turning angle correction,
- Δl, the telescope arm length correction,
- $\Delta\beta$, the gripper angle correction.

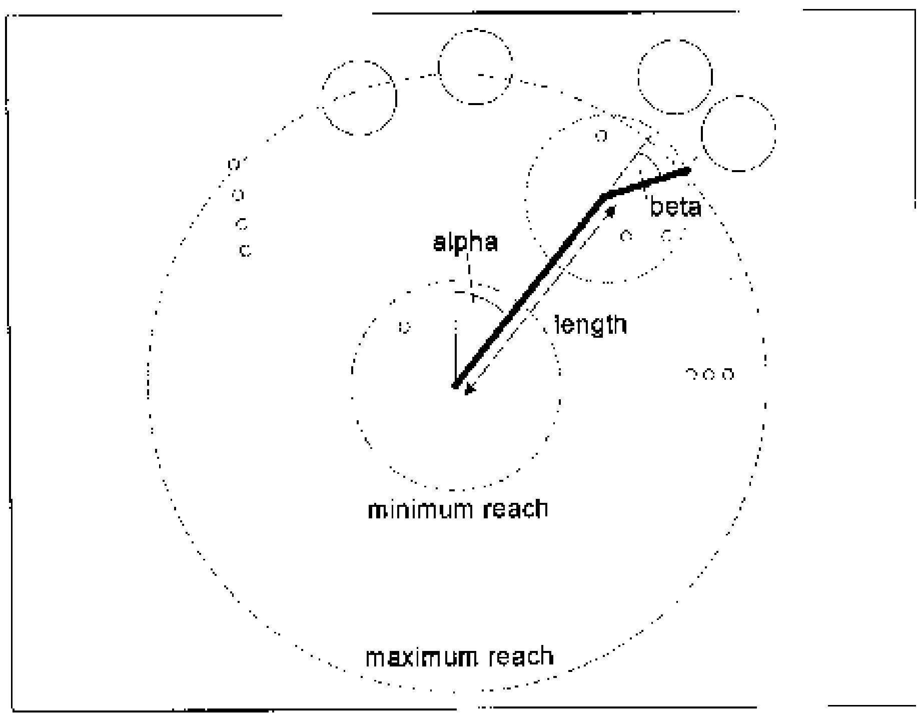


Figure44. Grasping Parameters.

The inputs to the fuzzy controller are (Figure , Figure, and Figure):

- *base_fl, base_fr*: the maximum angle or rotation of the base to the left and right without colliding.
- *base_t*: the angle of base rotation to the target.

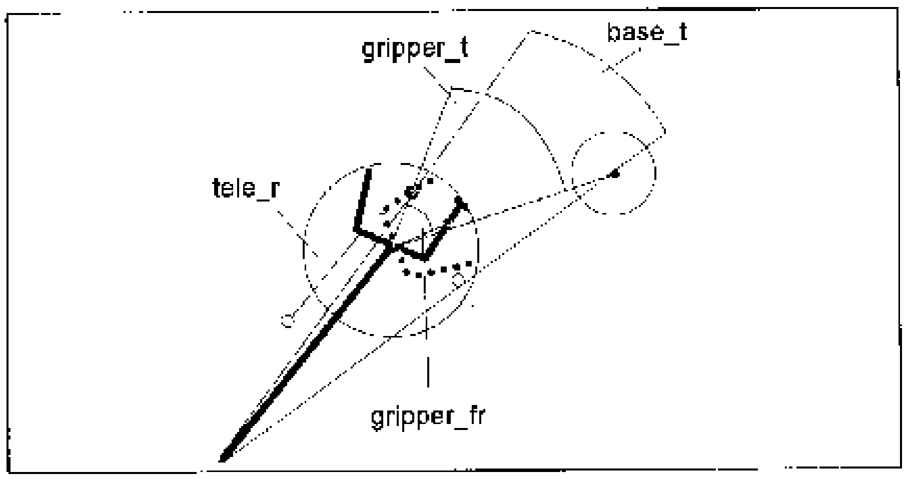


Figure 45. Input Parameters: gripper_t, base_t, tele_r, gripper_fr.

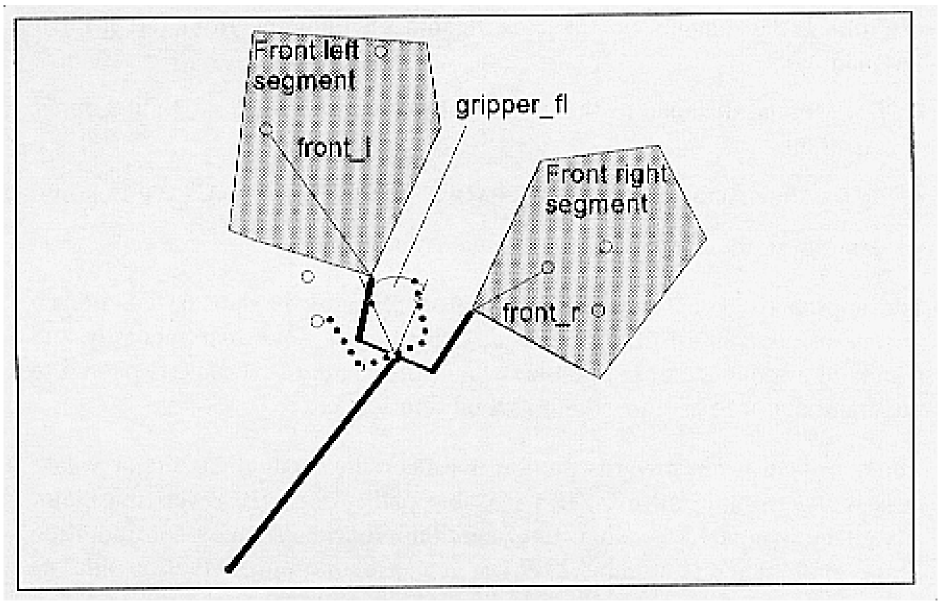


Figure 46. Input Parameters: front_l, front_r, gripper_fl

- *tele_e, tele_r*: the maximum extraction and retraction of the telescope arm without colliding.
- *tele_t*: the amount of extraction/retraction to the target.
- *gripper_fl, gripper_fr*: the maximum angle or rotation of the gripper to the left and right without colliding.
- *gripper_t*: the angle of gripper rotation to the target.

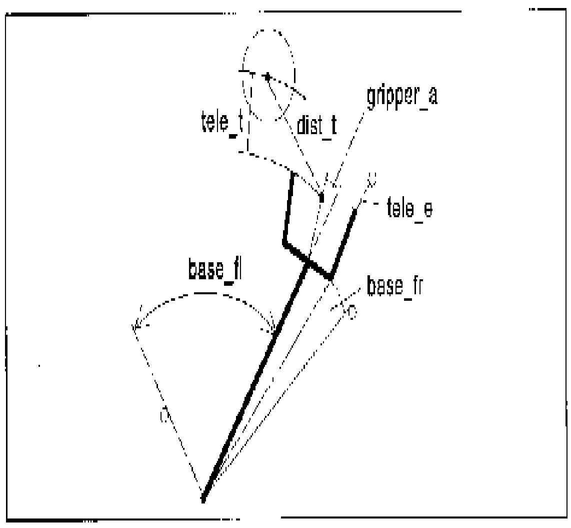


Figure 47. Input Parameters: gripper_a, tele_e, base_frm base_fl, tele_t, dist_t

- *front_l*: the distance to the closest object within the front left gripper quadrant.
- *front_r*: the distance to the closest object within the front right gripper quadrant.
- *dist_t*: the distance of the tool reference point to the target's centre point.
- *gripper_a*: the absolute gripper angle from the arm.

The approach taken for grasp and motion planning is state based, i.e. no history or analysis of effects is made. Planning is done incrementally and results in a sequence of tag points, which, if planning succeeds, is passed to the manipulator to be carried out as a motion sequence.

The approach taken towards motion is a heuristic method. Its major weakness is that it may fail even if a feasable path exists. However, the major advantage compared to exhaustive search approaches such as configuration space approaches (Latombe 1991) is its ease of implementation and its rather fast operation. On a 486-50 Mhz PC, planning takes about 2 to 5 seconds and is neglectable compared with execution of a typical path and grasp sequence which takes about 20 - 60 seconds.

Considering the application, grasping paper rolls from flat bed train cars, a certain amount of user intervention is tolerable. Typical cases of operator intervention are tightly stacked paper rolls which involve pushing and dragging as well as operation very close to obstacles. This is partly due to tolerances in locating paper rolls by the laser range finders, limited accuracy of physical movements, and incremental planning in discrete steps. All these inaccuracies and tolerances add up and cause the planner to fail under certain circumstances. A more precise path could be created by increasing the number of tag points, but that would increase planning time.

5.2.2 Hierarchical Fuzzy Rule Base

Traditional fuzzy control systems, which map fuzzy variables directly to output variables, are usually restricted to low-level control tasks, with a low number of input and output variables. Typical applications of fuzzy controller for low-level control tasks are the inverted pendulum, trailer backup tasks, car parking (Kosko 1992), and an anti-slip controller for vehicles (Akey 1995), (Hasemann et al. 1994). Complex behaviours, such as grasp and motion planning demand more sophistication.

During recent years several new approaches applying fuzzy logic to more complex tasks involving large numbers of inputs and outputs have been proposed. Among these is the ground breaking work of Saffioti et al. (1993a,b), who introduces fuzzy meta-rules for context determination, rule applicability, desirability, and the combination thereof to determine activation levels for lower level behaviours.

Nianzu et al. (1994) employ fuzzy control for robotic arm positioning. Voudouris et al. (1994) propose Fuzzy Hierarchical Control for autonomous vehicles with a more generalised fuzzy decision tree algorithm (Chang and Pavlidis 1977). This approach is based on fuzzy decision trees, which are analogous to classical decision trees but employ fuzzy meta rules on their nodes to determine the activation of subordinate nodes. The leaves of a fuzzy decision tree apply fuzzy rules to determine (after defuzzification) the outputs of the control system.

The approach taken by Voudouris et al. modifies low-level behaviours, i.e. traditional fuzzy systems incorporating weights (also known as Fuzzy Associative Memories, Kosko 1992) which map inputs to output values, by scaling the weights using fuzzy meta-rules.

Our approach of modelling complex behaviours using fuzzy logic stays with the original basic model of fuzzy inference using fuzzy rules without weights. Arbitration among different behaviours is done by adding “macros” (Figure 5), roughly corresponding to activation levels within Voudouris’s approach, to the antecedent of fuzzy rules (Figure 6). This is accomplished by conjunctively combining them with the orignal antecedent.

```
Macro definitions: (defining Brooks-like behaviours)

(macro on_target
   (is base_t target_close)
)

(macro approach_target
   (and
      on_target
      (is front_l o_medfar)
      (is front_r o_medfar)
      (is tele_e o_medfar)
      approach
   )
)

(macro fine_motion
  (not approach_target)
)
```

Figure 4. "Macros" defining Behaviour Contexts.

The major difference between our approach and Fuzzy Hierarchical Control lies in the effects of activation. Whereas in Fuzzy Hierarchical Control the output of each rule is scaled by the activation level, within our approach the output of each rule is limited by the activation level due to the min-max inference rule employed. Although “scaling” may result in smoother transitions when switching from one behaviour to another, we believe the possible performance gain over our “limiting” approach is minimal, although this remains to be researched.

```
Rules: (defining the controller output)

(rule
   (if
      (and
         fine_motion
         (is base_fl a_closefar)
         (is base_t target_left_close)
         (not (is front_l o_close))
      )
    then (is base turn_left_slow)
   )
)
```

Figure 5. Fuzzy Rule from the Fine Motion Rule Set.

The reason for using the “limiting” approach was its smooth integration into an existing fuzzy logic reasoning system. Moreover, our limiting approach is computationally cheaper because it adds only one min-operation (due to the conjunction of the macro statement with the antecedent of the rule) as opposed to at least one floating point multiplication in the “scaling” approach (multiplication of rule weight with the scaling factor).

Within the path and grasp planning system for the paper roll manipulator, the fuzzy logic reasoning system accounts for roughly 10 - 20% of the computational resources during planning, whereas geometrical reasoning and computation of input parameters accounts for the remaing 80 - 90%.

```
Fuzzy subsets defined as trapezoids over (-999, 999)

(subset turn_right_fast       2.0   5.0    6.0  8.0)
(subset turn_right_medium     0.5   1.0     1.5   2.0)
(subset turn_right_slow       0.0   0.1     0.5   1.0)
(subset turn_left_fast       -8.0   -6.0    -5.0  -2.0)
(subset turn_left_medium     -2.0  -1.5    -1.0  -0.5)
(subset turn_left_slow       -1.0  -0.5    -0.1   0.0)
(subset dont_turn            -0.05  0.00    0.00  0.05)

(subset extract_fast          0.3   0.4     0.5   0.6)
(subset extract_medium        0.2   0.3     0.4   0.5)
(subset extract_slow          0.0   0.05    0.1   0.3)

(subset retract_fast         -1.1  -0.7  -0.5  -0.4)
(subset retract_medium       -0.5  -0.4  -0.3  -0.2)
(subset retract_slow         -0.3  -0.1  -0.05  0.0)
(subset dont_move            -0.05  0.00  0.00  0.05)
```

Figure 6. Definition of Fuzzy Subsets as Trapezoids.

Within the motion and grasp planning for the paper roll manipulator the task of planning is conceptually decomposed into several rule sets (behaviours). Each rule set consists of one or more fuzzy rules (Figure 5). Fuzzy rules are antecedent precedent pairs.

Antecedents of fuzzy rules may be arbitrary well formed formulas over macros and (IS-variable-fuzzy-set)-expressions over input variables and fuzzy subsets. Precedents of fuzzy rules are output variable - fuzzy subset pairs. Fuzzy subsets are defined as trapezoidal membership functions (Figure 6). Default rules can be defined for output variables and are used if no other rule(s) can be applied (Figure 8).

```
Default rules:

(rule (default base 0))
(rule (default tele 0))
(rule (default gripper 0))
```

Figure 7. Default Rules.

Figure shows the organisation of different behaviours. The grasp and motion planning is divided into *Preapproach, Approach, Fine Motion,* and *Default Action* behaviours. *Preapproach* is activated if the distance to travel is large and the gripper is not in its home position (gripper angle = 0). The task of *Preapproach* is to turn the gripper angle into home position and to carefully retract the gripper. *Approach* is the behaviour of retracting the gripper and turning the base towards the target. *Left Approach* and *Right Approach* as well as *Left Approach Blocked* and *Left Approach Blocked* are sub-behaviours for obstacle avoidance during the ap-

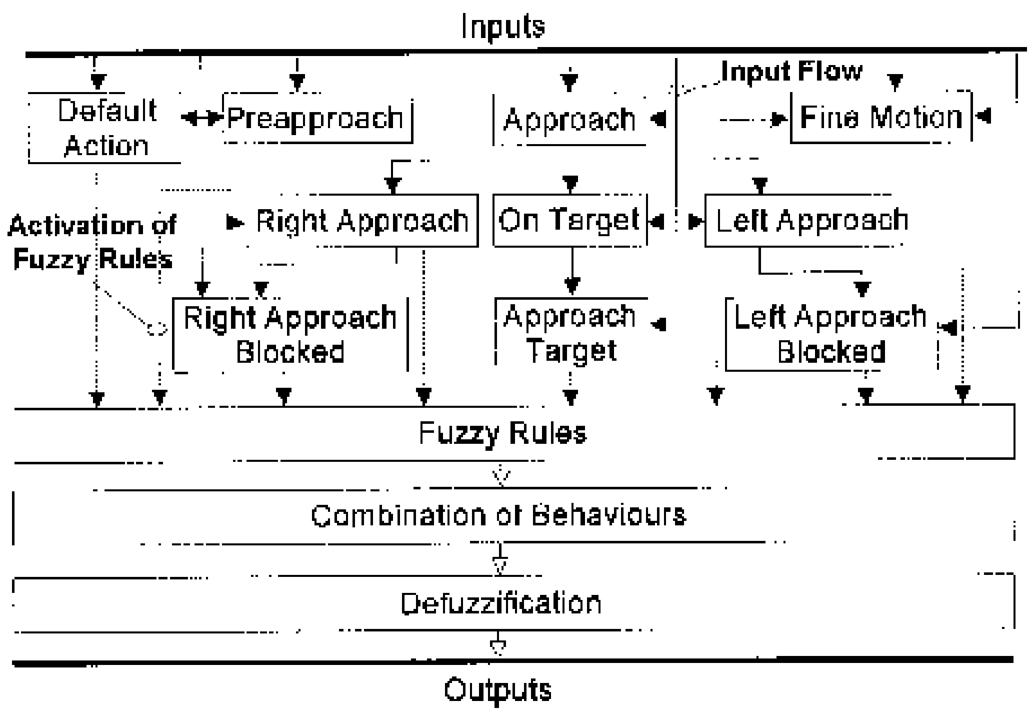
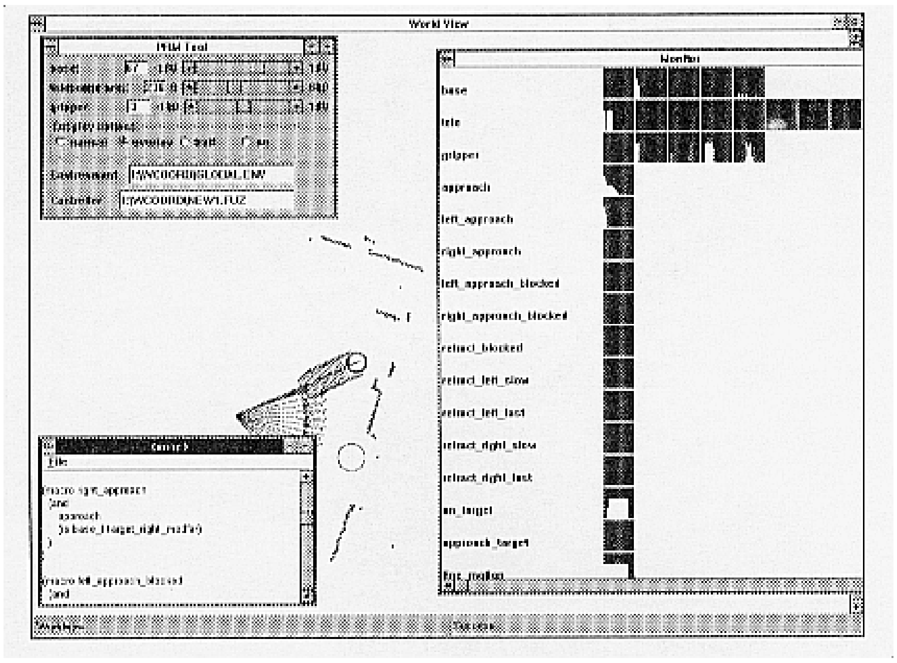


Figure52. Organisation of Planning Behaviours.

proach phase. In Figure *Left Approach Blocked* turns the telescopic arm towards the target paper roll whilst avoiding the obstacles to the left. *On target* implies that the base is directed towards the target and initiates extension of the telescopic arm. *Fine Motion* is initiated in close vicinity of the target and activates fuzzy rules responsible for careful approach to the target, the avoidance of neighbouring paper rolls, and grasping as depicted in Figure .

A Microsoft Windows 3.1 simulation environment has been implemented to develop and simulate the grasp and motion planning behaviour (Figure). Using this development environment, fuzzy controllers can be easily developed and tested. The activation potential of every rule and macro in the fuzzy controller definition file is shown at every control step (Monitor Window). Environments can be loaded, modified, and saved. The environment shown in Figure was generated from depth images obtained from the laser range scanner of a real environment.



Environment: I:\WCOORD\GLOBAL.ENV

Controller: I:\WCOORD\NEW1.FUZ

```
(macro right_approach
 (and
   approach
   (is base_f target_right_mid?)
 )

(macro left_approach_blocked
 (and
```

Figure53. Development and Simulation Environment

Trajectories can be recorded and saved as Windows Metafile Graphics for later analysis and documentation as in Figure and Figure .

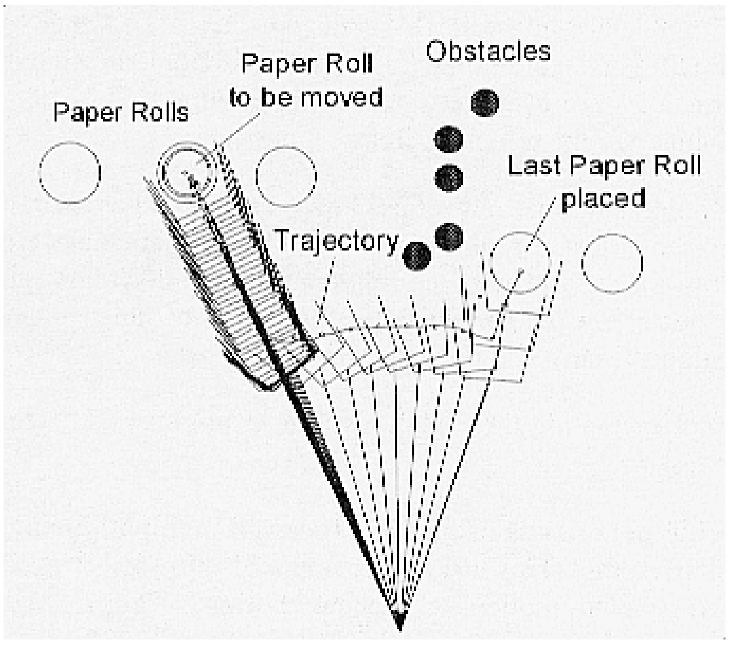


Figure 54. Grasping a Paper Roll (straight grasp).

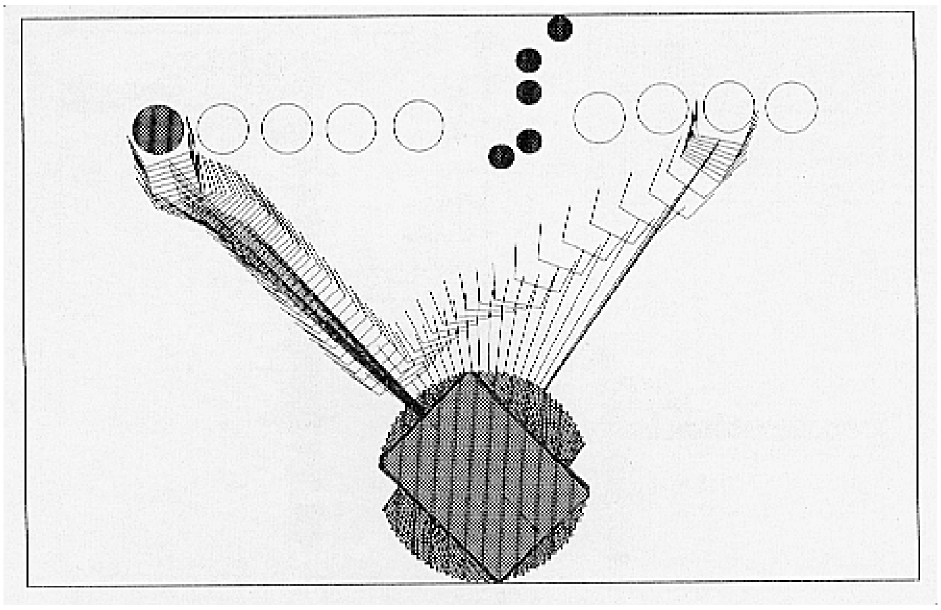


Figure 55. Grasping a Paper Roll (tilted wrist).

5.3 TASK SPECIFICATION

The implementation of the application scenario concentrates on the symbolic level. However, in order to demonstrate the integration of low level behaviours, the paper roll manipulator’s task of grasping and placing paper rolls has been implemented in much detail (Chapter 5.2). Other low level tasks such as path planning and moving for the autonomous guided vehicle as well as refueling have been very much simplified, e.g. refueling is done by simple driving over the refueling area.

The proposed robot control architecture, however, to a large extent concerns symbolic representations, e.g. the representation and maintenance of intentions. Therefore the lack of detailed implementations of atomic plan components does not affect the demonstrative character of the implementation of the application scenario and the test runs.

Within our implementation three high-level tasks are created. Figure illustates these tasks:

- **Transferring paper rolls using the paper roll manipulator** train car to a nearby buffer area. This task is decomposed into “select paper roll to be grasped”, “plan motion”, “execute motion”, “grasp paper roll”, “retract arm”, “select target location”, “plan motion”, “execute motion”, “release paper roll”, and “retract arm”.

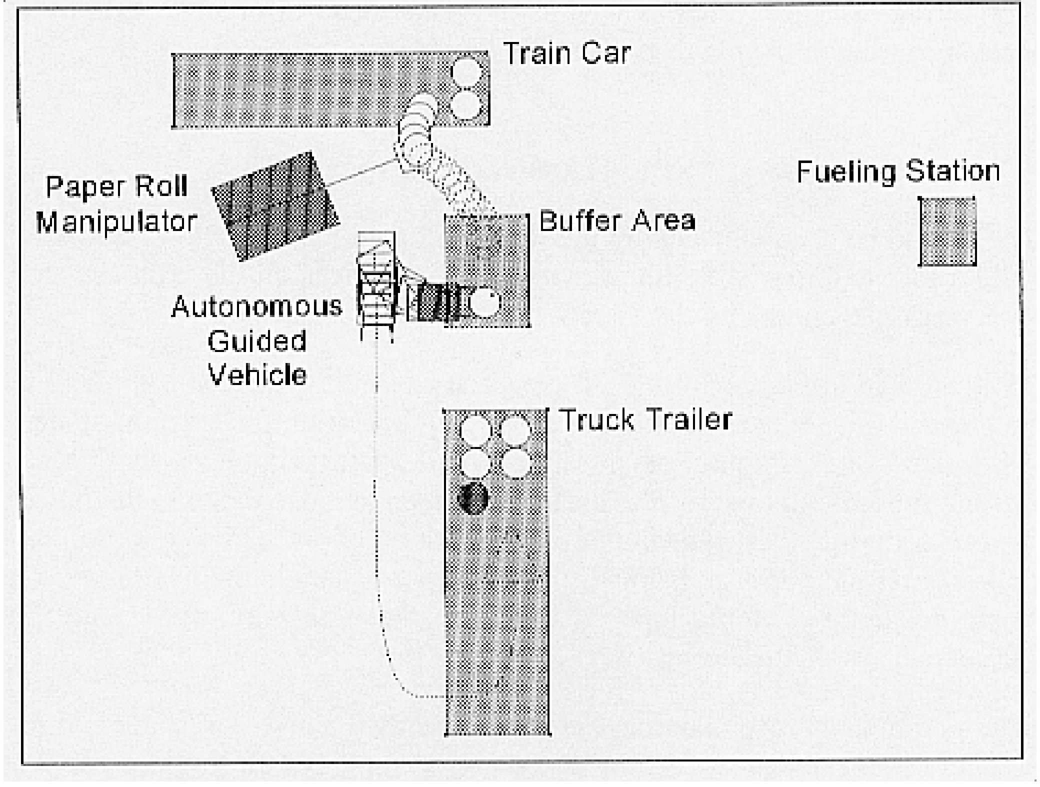


Figure 56. Material Flow Scenario.

- **Transferring paper rolls using the autonomous guided vehicle** from the buffer area to the trailer. Similar to the paper roll manipulator the task of the autonomous guided vehicle is decomposed into “select paper roll”, “drive to paper roll”, “grasp paper roll”, “drive to buffer”, “release paper roll”, and “drive to home position”. Drive commands are further decomposed into sequences of waypoints describing a path from the actual position to the destination. A limited number of alternatives to get from one position to another is described.

- **Refueling the autonomous guided vehicle**. This simple task is decomposed into “drive to fueling station” and “return from fueling station”. Refueling is assumed to take place while the AGV is driving inside the fueling station.

Aside from physical plan components, virtual plan components are used to further constrain physical plan component execution, decomposition, and intention switching. The paper roll manipulator task description for transferring paper rolls from the buffer area to the train car includes virtual plan components for monitoring the buffer area in order to avoid interference with the AGV while grasping paper rolls.

The AGV task description for transferring paper rolls from the trailer to the buffer area as well as for refueling includes virtual plan components for monitoring the buffer area as well as the path ahead in order to avoid objects appearing in the planned course of the AGV.

5.3.1 PRM: Transferring Paper Rolls

The task to be accomplished by the paper roll manipulator is to grasp paper rolls from the buffer area, transfer them to free places on the train car, and vice versa (depending on the direction of the material flow).

As illustrated in Figure this job comprises selecting the best paper roll to be grasped, planning the motion and grasping operation for grasping it, carrying them out, gripping the paper roll, and again planning a motion sequence for moving towards the buffer area, then in close range to the buffer area selecting the best target site, planning the motion sequence to do that, and placing the paper roll. Finally, the paper roll manipulator returns to a home position, i.e. the gripper is turned to the 0-degree-position and the telescopic arm is fully retracted.

The motion and grasp planning has been described above. Grasping and releasing paper rolls is carried out by simply opening and closing the gripper. This corresponds with the real behaviour of the paper roll manipulator which uses an intelligent gripper with slip sensors which minimizes the pressure applied on the paper rolls while avoiding slippage of the paper rolls within the gripper.

The activities of the paper roll manipulator are constrained by virtual plan components, e.g. to ensure that significant parts of the work environment are unchanged during operation.

5.3.2 AGV: Transferring Paper Rolls

The AGV’s task is to transfer paper rolls from a buffer area to a trailer nearby and vice versa (depending on the direction of the material flow). During these transfer tasks the fuel level should be kept at a reasonable level.

The task of transfering paper rolls involves moving from the current position to the buffer area, selecting the most appropriate paper roll, gripping the paper roll, backing up, driving to the trailer, selecting the best target location, placing the paper roll, and returning to the home position.

Navigating and moving through the environment is done in a hierarchical manner. When driving from one location to another a skeletal plan consisting of a sequence of tag points is created which is refined throughout the course of action.

Constraints for the task of transferring paper rolls are accessibility of the buffer area and no obstacles on the path to be driven.

5.3.3 AGV: Refueling

The plan for refueling is simply driving to the fueling station and remaining there for the time of refueling. A person at the station (or perhaps another robot) performs the physical refueling. Although simple in itself, the engagement and disengagement of this task is rather tricky, i.e. deciding when to refuel and where to return. Constraints are the accessibility of the fueling station and no obstacles on the path to be driven.

5.4 TASK INITIATION

Tasks are initiated by intention switching, i.e. whenever the associated trigger functions return TRUE. Intention switching is done by inserting the abstract plan component representing the task into the current execution graph. Insertion of a plan component takes into account virtual plan components the current execution graph and checks their compatibility with decomposition constraints attached to the abstract plan component to be inserted.

A set of possible insertion points is returned as a set of ordering constraints. A heuristic attached to the plan component selects the most suitable (e.g. distance to the fuel depot vs. current fuel level).

Figure illustrates the changes in the current execution graph (some plan components removed for clarity). Within this current execution graph a grip-transfer-release sequence for the AGV is currently planned. The plan components “1” through “9” denote move commands for different parts of the way from the trailer to the buffer area. In this situation the refueling could have been inserted at three

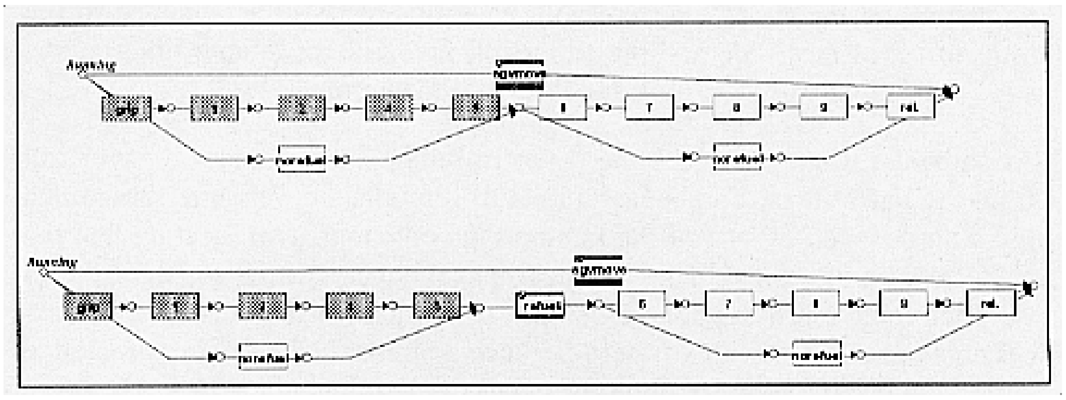


Figure 57. Initiation of Refueling by Plan Component Insertion.

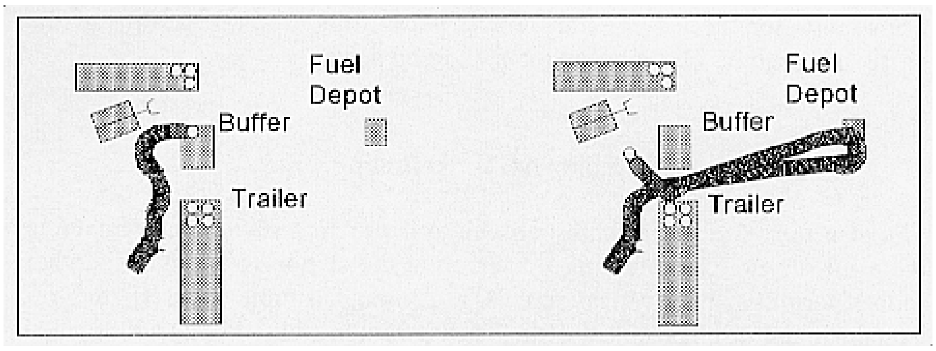


Figure 58. "Refueling" (right) by Plan Insertion into "Transport" (left).

different places (as constrained by the “no refueling” constraints): at the beginning, the end, or the chosen node between “5” and “6”. The third place was selected in this example.

Figure (left) shows the normal behaviour, i.e. transport paper rolls, and what happens (right) when refueling becomes necessary while the AGV is planned to transport a paper roll from the trailer to the buffer. The AGV starts with “transport”, proceeds through several of motion primitives (“1” - “5” in Figure), and then decomposes and executes the “refueling” behaviour which returns the AGV to the point from which it can continue the “transport” behaviour.

Task execution is controlled by the sequentiality constraints represented by the graph structure. Fallback occurs when a decomposition has been completely executed.

5.5 FAULT RECOVERY

Two strategies are used for fault recovery. The most common one is plan modification. In Figure the AGV and the paper roll manipulator cooperate without explicit coordination of their activities. Consequently, every now and then situations occur in which the operations of the paper roll manipulator and the AGV interfere with each other. The tags attached to the trajectories of the AGV and the paper roll manipulator denote time points, e.g. “t1” denotes the start time and “t4” the the end time.

As shown, at time point “t2” the paper roll manipulator grasps a paper roll from the buffer area. The behaviour controlling the AGV is interrupted due to an alert issued by one of the motion plan components indicating that the buffer area is occupied. This initiates fault recovery which for the sake of simplicity and clarity is done by simply waiting until the buffer area is accessible again (in fact this is safe because a similar mechanism is included within the behaviour controlling the paper roll manipulator).

Obviously, “waiting” could be achieved in two ways, (a) by replanning the motion primitive as “wait” and then “move”, or (b) via plan component insertion by insertion of a new plan component “wait for buffer area free” right in front of the concerned plan component. Whereas (a) is computationally more efficient and chosen here for illustration (b) would be advantageous if the designer wants to detect deadlocks.

Fault recovery depends on fault detection by virtual plan components. Whenever a virtual plan component detects a deviation from the planned course of actions, an error message is raised. Error messages include information about the origin and type of the associated error.

As in TCA (Simmons 1994) an exception handler is sought to take care of recovery actions. Global fault recovery is accomplished by global exception handlers whereas local fault recovery is attempted by local exception handlers which are attached to abstract plan components.

The search for a suitable exception handler starts with the global exception handlers and continues (if none is found) with local exception handlers starting from the next higher level (with respect to the level in which the error occured). To carry out error recovery, all plan components on the current level or below are terminated (if currently executing) and removed. The exception handler of the next higher (if any) level is then tried for error recovery. If unsuccessful, the process of local error recovery successively continues to the next higher levels.

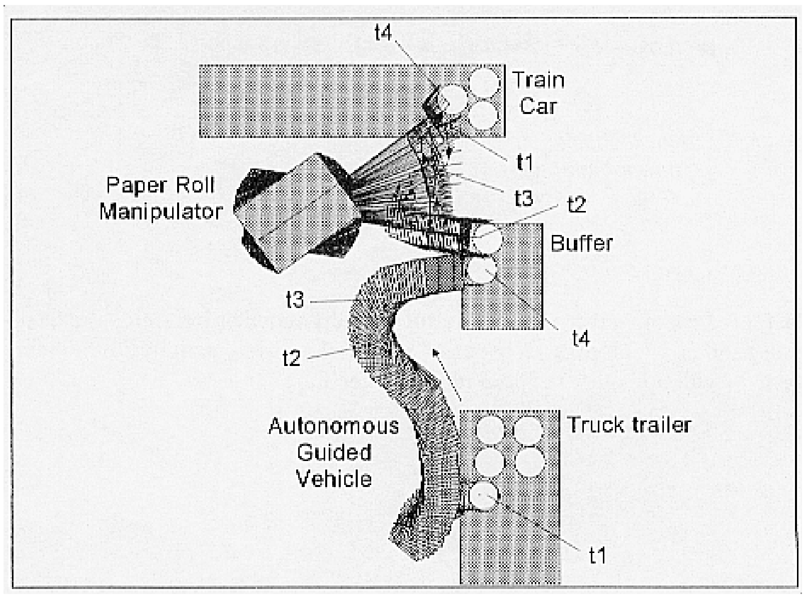


Figure 59. AGV waits for access to the buffer area.

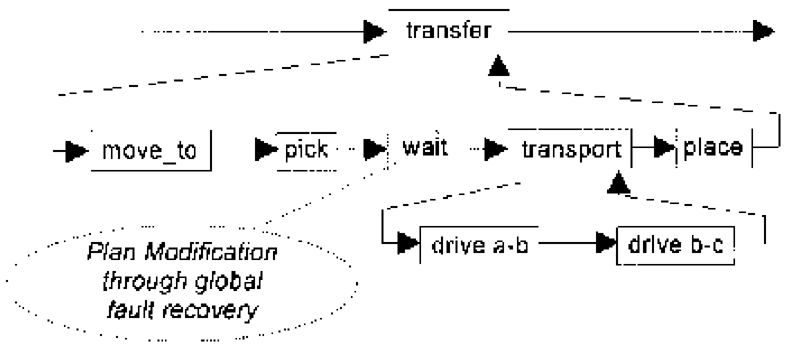


Figure 60. An error occurs during execution of "drive a-b".

The following example will illustrate the fault recovery concepts in detail. Figure illustrates the situation in which an error occurs during the execution of a drive command. The incident that the intermediate way point “b” is blocked is indicated by an error message.

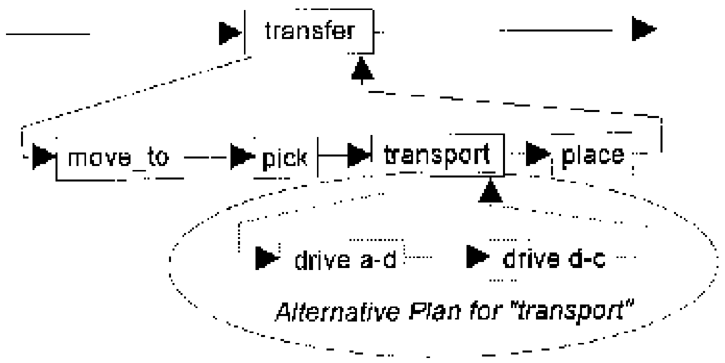


Figure 61. Global error recovery.

In Figure this error message is successfully handled by a global exception handler which tries recovery by inserting a “wait” activity in the hope that “b” will no longer be blocked after some time.

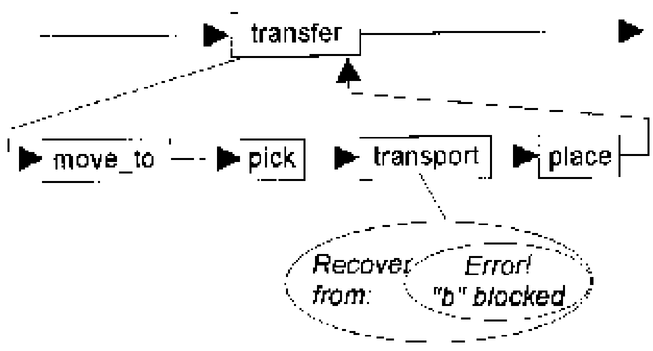


Figure62. Local error recovery by falling back to the next higher level.

In Figure local error recovery is illustrated. The error that occured in Figure is this time addressed by falling back to the next higher level of abstraction. The exception handler tied to the plan component “transport” is then run to plan recovery from the fault.

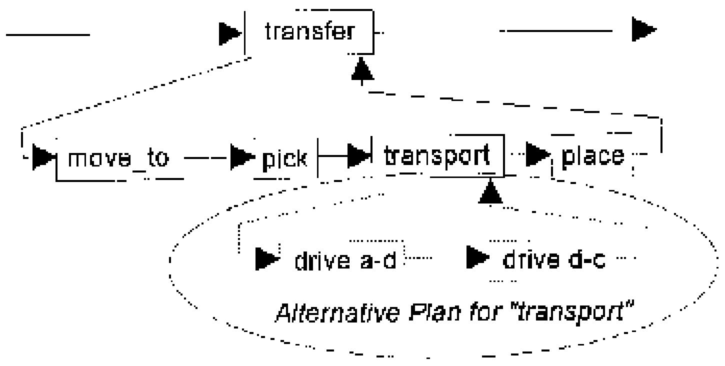


Figure 63. Local error recovery succeeds by finding an alternative plan.

Figure depicts the case in which the exception handler associated with “transport” is able to create an alternative plan.

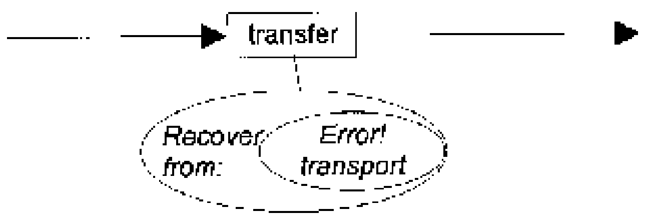


Figure 64. Local error recovery fails to supply an alternative plan.

In Figure the unsuccessful case is shown, i.e. the exception handler for “transfer” fails to supply an alternative plan and fallback continues to the next higher level. During fallback the original error message “‘b’ blocked” is translated into a “transport” error message. Due to this translation the error message is on the same level of abstraction as the currently concerned exception handler. “Abstracting” error messages is an important concept to hide implementational details of lower levels from more abstract levels.

Fatal Error!
transfer failed!

Figure 65. No error recovery was found.

If all error recovery attempts fail, the system returns a fatal error message as shown in Figure .

5.6 CONTROL SYSTEM ENGINEERING

Although the experience gained through experiments and trial runs with our proposed robot control architecture is limited to the described implementation tested in a simulated environment, a few insights were gained on how to develop control systems for intelligent robots applying the proposed robot control architecture.

We believe control system engineering has to start with the design of low-level behaviours and the mapping of them to plan component classes. This way, a tool box of low-level plan components is created and can be tested on simple tasks. During testing, exceptional conditions are determined, which in the future will constrain the applicability of low-level operators.

The next step is to arrange simple operators into task nets to form higher level abstact plan components. During this step the applicability of plan components must be constrained to reasonable situations. Further testing reveals further exceptional conditions with respect to the applicability of certain task nets. This information can then be used to constrain the applicability of task nets or for further design of recovery activities.

We believe that the design of a control system for intelligent robots should be an incremental bottom-up approach which employs simulations and real world tests for further information on the applicability of plan components and for the development of recovery strategies.

The implementation of the proposed robot control architecture in an object-oriented language draws strong benefits from the notion of plan component types which can be easily implemented as base classes of application dependent plan components which (merely) override default functionalities with application dependent code. Figure 10 shows an excerpt of a C++ header file defining application specific plan components. “Prmmove” and “prmpickplace” are abstract plan components and descendants of a base class “abstract_pc”. “Prmmove” is at the same time a behaviour, in the sense that it can be activated under certain circumstances (intention switching). Hence, it overrides the trigger function of “abstract_pc” with application specific code. In the same fashion “prmplan” as an atomic plan component gets its default functionalities from the base class “atomic_pc”.

```
class prmmove : public abstract_pc
  {
  public:
      int decompose(void);
      int trigger_function(void);
      prmmove(int f, int t, int i, char* n);
      prmmove* get_insert_action(void);
      int get_constraints(my_list<constraint>* l);
  };

class prmpickplace : public abstract_pc
  {
  public:
      int decompose(void);
      prmpickplace(int f, int t, int i, char* n);
  };

class prmplan : public atomic_pc
  {
  private:
     struct prm_configuration prm_work;
     struct prm_configuration prm_save;
  public:
      int execute(void);
      prmplan(int f, int t, int i, char* n);
      prmplan* copy(void);
  };
```

Figure 8. Application Specific Plan Components are Descendants of Base Classes.

5.7 DISCUSSION

Within the implemented application scenario the number of implemented high-level tasks may seem small. We believe, however, that intelligent robots targetting the service market will not exhibit much more flexibility in near future because increased flexibility goes along with increased sophistication of mechanical structures and increased number of actuators and sensors which increases costs. For example the Fraunhofer Institute (FhG-IPA 1994b) proposes an intelligent robot is proposed for cleaning sanitary installations. In this proposal only a subset of cleaning tasks is done by the robot leaving a significant amount of work to be done by cleaning personnel. Subsequent calculations based on an assumption of 33% savings of personnel costs and a depreciation over four years suggest a target price of DM 30.000 ($ 20.000). This target price implies the use of simple sensors, actuators, and structures and thus rather simple tasks.

Moreover, the introduction of intelligent robots will be gradual both in numbers and the extent of automation. Full automation of many service tasks, at least in near future, is neither commercially feasible nor socially acceptable. Consequently, we expect so called service robots to concentrate on automating the most time consuming tasks leaving others to human operators. This concentration is reflected by a relative small number of high-level tasks.

Another point of critique might be why our example includes two robots, the autonomous guided vehicle and the paper roll manipulator, whereas the robot control architecture is one for a single intelligent robot. The reason we chose to control two robots was simply to increase complexity and the possibility of interactions between the robots which mutually affect each other's working environment. We believe, that this approximates the dynamics a single intelligent robot of the next generation will be faced with in the real world. Further disturbances can, however, be introduced in the simulation environment, e.g. by placing and moving objects, e.g. paper rolls and obstacles.

In our architecture, the biggest practical issue during the design of robot control systems is the need for tool support for designing task nets for decomposition. Whereas the definition of heuristics is well supported using fuzzy logic in more complicated cases or plain C language code in simple cases, the definition of subplans is currently very cumbersome.

To support the development, a graphical tool could be envisaged which allows the designer to define decompositions by drag and drop from an existing library of plan components and arranging them on a graphical workbench. Heuristics defining their applicability or criticality could be attached to abstract plan components as minispecs either as fuzzy rule base or as C language code. The tool could then compile the graphical representation into C++ code which in turn is linked to robot control architecture code.

Another issue is the design of an operating system dependent scheduler supporting the proposed robot control architecture. Schedulability of a plan component itself is an important issue. For real world applications it must be ensured that time critical routines and task such as servo loops and monitoring requests meet their timing requirements and that modifications within the current execution graph only occur during non-critical intervals.

From observation there seems to be no black and white distinction between normal and exceptional execution but rather a continuum with normal and excep tional execution as the extremes. Fault recovery on various level of abstraction causes the blurring of notions of normal and exceptional operation.

This poses the question where to draw the line between the normal run of execution and fault recovery, how much effort should be spent to get the task done right in first place as opposed to trying reasonably promising strategies with the risk of failing and going through fault recovery. Trying to make the nominal execution more reliable results in much higher efforts in selecting the right approach for a certain task and may also result in inefficiencies when, due to uncertainties less, risky but relatively inefficient decompositions are chosen.

Within the simulation examples two choices exist: (a) to coordinate the access to the buffer area, guaranteeing that access is mutually exclusive or (b) no coordination, using fault recovery in cases when access to the buffer area is not possible (e.g. by waiting until the buffer area is free when it is temporally occupied by the AGV or the paper roll manipulator). Whereas the first approach results in a much more complicated task description and inefficiencies during plan component insertion, the second approach is very simple with respect to the task description and the efforts spent on fault recovery, i.e. just waiting.

This is because plans for different intentions are not merged but are based on insertions of a single plan component which are successively decomposed. Hence, the two different intentions continue to exist as distinct entities within the current execution graph and are not intertwined. This has the advantage of simplicity but causes the execution of two intentions to be sequential if they are not completely independent from each other.

A possible workaround to this is to divide a complex behaviour into a set of simpler behaviours which are event triggered, e.g. for the paper roll manipulator in the simulation example this could be a “grasping” behaviour, a “moving to the buffer area” behaviour, a “placing behaviour”, and a “returning to home position” behaviour which are triggered on demand. This solution approximates the behaviour of knowledge areas originally proposed by Georgeff and Lansky (1987) and rule based control proposed by Matsushita et al. (1993).

6 CONCLUSIONS

In this thesis different planning and monitoring techniques are discussed and related to each other. Complexity results of planning are reviewed and monitoring in general and its importance within robot control architectures are discussed. This thesis extends the traditional view of monitoring towards pre-execution conflict detection. The difference between monitoring discrete event systems and continuous event systems is investigated and the integration of monitoring for mixed discrete and continuous event systems is analysed.

Newly developed methods of representing control knowledge as plans and the process of plan expansion are discussed. Within the chosen approach plans are represented as graphs over plan components. Plan components explicitly represent atomic and abstract activities as well as dependencies, relationships, and constraints. The plan component life cycle for physical plan components is described.

New monitoring concepts are presented, introducing plan execution and plan validity monitoring as well as the automatic detection of causal links from STRIPS-like plans. Results achieved from an implementation of these concepts using the monitoring shell JANUS are briefly reviewed.

A planning mechanism using the above structures is proposed. This planning mechanism is based on graph manipulation and has similarities to Hierarchical Task Network planning systems. This method is both used to describe the dynamics of plan execution and for fallback in case of error.

Another contribution lies in the proposed intention switching mechanism. Intention switching is context dependent and allows the control system to coordinate and accomplish conceptually different high-level tasks (intentions). Intentions are modelled by abstract plan components with trigger functions defined to continuously compute the criticality of each intention depending on the current context. Intention switching is initiated if the criticality exceeds a certain threshold and is carried out by finding a suitable insertion point in the current execution graph and inserting the abstract plan component, i.e. the triggered intention, there. Successive decomposition and execution yields the desired behaviour.

Moreover, a formal basis for monitoring plan execution and plan validity is elaborated and implementation aspects are discussed.

An implementation of the robot control architecture has been carried out and the results and experience have been reported. Aside from a cumbersome definition of decompositions, the implementation of the robot control architecture turned out to be straightforward to implement (aside from the scheduler) largely due its inherent object-oriented design associating all activities to different but similar classes of plan components. The robot control architecture has been applied to a fairly complex simulated environment with different sources of disturbances and interactions.

The implementation of complex low-level behaviours using a new model of hierarchical fuzzy logic rule bases has been demonstrated and is used for grasp and motion planning within both the simulation environment and an industrial paper roll manipulator.

7 REFERENCES

Aaltonen, K. 1993. Palvelurobotiikan nykytila ja sovellukset. MSc Thesis. Tampere Technical University, Department of Mechanical Engineering, Tampere, Finland. 98 pp. (In Finnish.)

ACS (ACS-Technologies) 1995. C-PRS - Technical Representation. ACS Technologies, Labege, France. 26 pp.

Akey, M. L. 1995. Fuzzy Logic Anti-skid Control for Commercial Trucks. Proceedings of the SPIE Conference on Applications of Fuzzy Logic, Orlando, Florida, April 19 - 21, 1995. SPIE - The International Society for Optical Engineering, Bellingham, Washington. Pp. 359 - 370.

Albus, J. S. 1995. RCS: A Reference Model Architecture for Intelligent Systems. Proceedings of the 1995 AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents, Stanford University, March 27 - 29, 1995. AAAI/MIT Press, Boston, Massachusetts. Pp. 1 - 6. (Currently at ftp://hobbes.jsc.nasa.gov/pub/ korten/spring_symposium/ submissions)

Albus, J. S., McCain, H.G. & Lumia, R. 1989. NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM). NIST Technical Note 1235, 1989 Edition. National Institute of Standards and Technology, U.S. Dept. of Commerce. 76 pp.

Arkin, R. C. 1989. Dynamic Replanning for a Mobile Robot based on Internal Sensing. Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-89), Scottsdale, Arizona, May 14 - 19, 1989. IEEE Computer Society Press, Los Alamitos, California. Pp. 1416 - 1421.

Backström, C. 1992. Computational Complexity of Reasoning about Plans. Linköping University, Department of Computer and Information Science, Dissertation No. 281. Linköping, Sweden. 206 pp.

Benson, S. & Nilsson, N. 1995. Reacting, Planning and Learning in an Autonomous Agent. Machine Intelligence 14. Ed. by K. Furukawa, D. Michie, and S. Muggleton. Clarendon Press, Oxford. (Forthcoming.)

Bonasso, R. P. & Kortenkamp D. 1994. An Intelligent Agent Architecture in Which to Pursue Robot Learning. Working Notes: MCL-COLT '94 - Robot Learning Workshop, July, 1994. 7 pp.

Bonasso, R. P. & Kortenkamp D. 1995. Characterizing an Architecture for Intelligent, Reactive Agents. Proceedings of the 1995 AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents, Stanford University, March 27 - 29, 1995. AAAI/MIT Press, Boston, Massachusetts. Pp. 29 - 34. (Currently at ftp://hobbes.jsc.nasa.gov/ pub/korten/spring_symposium/submissions)

Bonasso, R. P. 1991. Integrating Reaction Plans and Layered Competences Through Synchronous Control. Proceedings of the International Joint Conference on Artificial Intelligence, Sydney, Australia, August 24 - 30. Pp. 1225 - 1231.

Brooks, R. A. 1989. A Robot that Walks; Emergent Behavior from a Carefully Evolved Network. Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-89), Scottsdale, Arizona, May 14 - 19, 1989. IEEE Computer Society Press, Los Alamitos, California. Pp. 692 - 696. Currently at ftp://publications.ai.mit.edu/ ai-publications/ 500-999/AIM-864.tiff.tar.gz

Bylander, T. 1992. Complexity Results for Extended Planning. Proceedings of the 1st International Conference on AI Planning Systems, College Park, Maryland, June 15 - 17, 1992. Morgan Kaufmann Publishers, San Mateo, California. Pp. 20 - 27.

Chang, R. L. P. & Pavlidis, T. 1977. Fuzzy Decision Tree Algorithms. IEEE Transactions on Systems, Man, and Cybernetics, Vol. 7, No. 1, pp. 28 - 35.

Chapman, D. 1987. Planning for Conjunctive Goals. Artificial Intelligence, Vol. 32, pp. 333 - 378.

Chatila, R., Alami, R., Degallaix, B. & Laruelle, H. 1992. Integrated Planning and Execution Control of Autonomous Robot Actions. Proceedings of the 1992 IEEE International Conference on Robotics and Automation (ICRA-92), Nice, France, May 12 - 14, 1992. IEEE Computer Society Press, Los Alamitos, California. Pp. 2689 - 2696.

Connell, J. H. 1992. SSS: A Hybrid Architecture Applied to Robot Navigation. Proceedings of the 1992 IEEE International Conference on Robotics and Automation (ICRA-92), Nice, France, May 12 - 14, 1992. IEEE Computer Society Press, Los Alamitos, California. Pp. 2719 - 2724.

Currie, K. & Tate, A. 1991. O-Plan: the Open Planning Architecture. Artificial Intelligence, Vol. 52, pp. 49 - 84.

Davis, E. 1992. Semantics for Tasks that can be Interrupted or Abandoned. Proceedings of the 1st International Conference on AI Planning Systems, College Park, Maryland, June 15 - 17, 1992. Morgan Kaufmann Publishers, San Mateo, California. Pp. 37 - 43.

Donald, B. R. 1987. Error Detection and Recovery in Robotics. Lecture Notes in Computer Science. G. Goos and J. Hartmanis (Eds.). Springer-Verlag. Berlin, Heidelberg, New York. 314 pp.

Dorn, J. 1989. Wissensbasierte Echtzeitplanung, Künstliche Intelligenz. Verlag Friedrich Vieweg & Sohn, Braunschweig/Wiesbaden, Germany. 167 pp. (In German.)

Dorn, J. 1994. Dependable Reactive Event-oriented Planning. Data & Knowledge Engineering, Vol. 16, pp. 27 - 49.

Doyle, R. J., Atkinson, D. J. & Doshi, R. S. 1986. Generating Perception Requests and Expectations to verify the Execution of Plans. Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86), Philadelphia, Pennsylvania, July 1986. AAAI Press, Menlo Park, California. Pp. 81 - 88.

Drummond, M. 1994. On Precondition Achievement and the Computational Economics of Automatic Planning. Proceedings of the EWSP'93 - 2nd European WorkShop on Planning. Vadstena, Sweden, December 9 - 11, 1993. Current Trends in AI Planning, C. Bäckström and E. Sandewall (Eds.). IOS Press, Amsterdam, Netherlands. Pp. 6 - 13. (Currently at http://www.electric-time.com/papers/ewsp-93.ps)

Drummond, M. E. & Kaelbling, L. P. 1990. Integrated Agent Architectures: Benchmark Tasks and Evaluation Metrics. Technical Report. Teleos Research, Palo Alto, California. 4 pp.

EC (European Commission Directorate-General III) 1994. IT R&D Programme, Computer-integrated Manufacturing and Engineering. Summaries of Esprit projects. Report EUR 15375, ECSC-EC-EAEC, Brussels, Belgium. 421 pp.

Erol, K., Hendler, J. & Nau, D. 1994a. Semantics for Hierarchical Task-Network Planning. University of Maryland, Computer Science Technical Report Series, CS-TR-3239. College Park, Maryland. 28 pp. (Currently at http://www.cs.umd.edu/~kutluhan/ Papers/htn-sem.ps)

Erol, K., Hendler, J. & Nau, D. 1994b. Complexity Results for HTN Planning. University of Maryland, Computer Science Technical Report Series, CS-TR-3240. College Park, Maryland. 30 pp. (Currently at http://www.cs.umd.edu/~kutluhan/Papers/AMAI.ps)

Ferguson, I. A. 1992. Toward an Architecture for Adaptive, Rational, Mobile Agents. Proceedings of the Third European Workshop on Modelling Automous Agents in a Multi-Agent World. Decentralized AI - Vol. 3. Kaiserslautern, Germany, 5 - 7 August 1991. E. Werner and Y. Demazeau (Eds.) Elsevier Science Publishers B.V. (North Holland), Amsterdam, Netherlands. Pp. 210 - 218.

Ferguson, I. A. 1995. Integrating Models and Behaviors in Autonomous Agents: Some Lessons Learned on Action Control. Proceedings of the 1995 AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents, Stanford University, March 27 - 29, 1995. AAAI/MIT Press, Boston, Massachusetts. Pp. 78 - 91. (Currently at ftp://hobbes.jsc.nasa.gov/pub/korten/spring_symposium/submissions)

FhG-IPA (Fraunhofer Institute for Manufacturing Engineering and Automation) 1994a. Innovationen für den Dienstleistungsbereich Unterhaltsreinigung. Bericht zum IPA-Technologie-Forum, Stuttgart. 70 pp. (In German.)

FhG-IPA (Fraunhofer Institute for Manufacturing Engineering and Automation) 1994b. Serviceroboter - ein Beitrag zur Innovation im Dienstleistungswesen, Eine Untersuchung des Fraunhofer-Instituts für Produktionstechnik und Automatisierung (IPA), Stuttgart. 62 pp. + 100 pp. (In German.)

Fikes, R. E. & Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence, Vol. 2, pp. 189 - 208.

Firby, J. 1987. An Investigation into Reactive Planning in Complex Domains, Proceedings of the Sixth National Conf. on Artificial Intelligence (AAAI-87), July 13 - 17, Seattle, Washington. AAAI Press, Menlo Park, California. Pp. 202 - 206

Firby, J. 1989. Adaptive Execution in Complex Dynamic Worlds. Ph.D. Thesis. Yale University Technical Report, YALEU/CSD/RR #672, Yale. 356 pp.

Firby, J. 1995. An Architecture for A Synthetic Vacuum Cleaner. Proceedings of the 1995 AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents, Stanford University, March 27 - 29, 1995. AAAI/MIT Press, Boston, Massachusetts. Pp. 97 - 111. (Currently at ftp: //hobbes. jsc.nasa.gov/pub/korten/ spring_symposium/submissions)

Flynn, A. & Brooks, R. 1989. Building Robots: Expectations and Experiences. Proceedings of the IEEE/RSJ International Workshop on Intelligent Robots and Systems (IROS'89). Tsukuba, Japan, Sept. 4 - 6. IEEE, Piscataway, New Jersey. Pp. 236 - 243.

Gat, E. 1991. Integrating Reaction and Planning in a Heterogeneous Asynchronous Architecture for Mobile Robot Navigation. SIGART Bulletin, Vol. 2, No. 4, 1991, pp. 70 - 74.

Gat, E. 1993. On the Role of Stored Internal State in the Control of Autonomous Mobile Robots. AI Magazine 14, Spring 1993, pp. 51 - 62.

Gat, E., Desai, R., Ivlev, J. L. & Miller, D. P. 1994. Behavior Control for Robotic Exploration of Planetary Surfaces. IEEE Transactions on Robotics and Automation, August 1994, pp. 490 - 503.

Georgeff, M. P. & Lansky, A. L. 1987. Reactive Reasoning and Planning. Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87). Seattle, Washington, July 13 - 17. AAAI Press, Menlo Park, California. Pp. 677 - 682.

Ghallab, M. & Laruelle, H. 1994. Representation and Control in IxTeT, a Temporal Planner. Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS'94). Chicago, Illinois, June 13 - 15, 1994. AAAI Press, Menlo Park, California. Pp. 61 - 67.

Ginsberg, M. L. & Smith, D. E. 1988. Reasoning about Actions I: A possible Worlds Approach. Artificial Intelligence, Vol. 35, No. 2, pp. 165 - 195.

Goldman, R. P. & Boddy, M. S. 1994. Conditional Linear Planning. Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS'94). Chicago, Illinois, June 13 - 15, 1994. AAAI Press, Menlo Park, California. Pp. 275 - 280.

Habel, A. & Kreowski, H.-J. 1987. On Context-free Graph Languages Generated by Edge Replacement. Theoretical Computer Science, Vol. 51, pp. 81 - 115.

Hasemann, J.-M. 1992. Monitoring in Intelligent Autonomous Robots. MSc Thesis. University of Bremen, Department of Computer Science, Bremen, Germany. 80 + 70 pp.

Hasemann, J.-M. 1994a. Planning, Behaviours, Decomposition, and Monitoring Using Graph Grammars and Fuzzy Logic. Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS'94). Chicago, Illinois, June 13 - 15, 1994. AAAI Press, Menlo Park, California. Pp. 275 - 280.

Hasemann, J.-M. 1994b. A Robot Control Architecture based on Graph Grammars and Fuzzy Logic. Proceedings of the IEEE/RSJ/GI Intelligent Robots and Systems Conference (IROS'94). München, Germany, September 12 - 16, 1994. IEEE, Piscataway, New Jersey. Pp. 2123 - 2130.

Hasemann, J.-M. 1994c. Reactive Planning as Graph Rewriting. Proceedings of the 5th International Workshop on Graph Grammars and their Application to Computer Science (GRAGRA'94), Williamsburg, USA. Pp. 61 - 66.

Hasemann, J.-M. 1994d. Planning and Monitoring in Dynamic Environments. Licentiate Thesis. University of Oulu, Department of Electrical Engineering, Oulu, Finland. 101 pp.

Hasemann, J.-M. 1995a. A Control Architecture for Service Robots. Proceedings of the Fourth Golden West Conference on Intelligent Systems (GWICS'95). San Francisco, California, June 12 - 14, 1995. The International Society for Computers and their Applications - ISCA. Raleigh, North Carolina. Pp. 199 - 203.

Hasemann, J.-M. 1995b. Robot control architectures - Application requirements, approaches, and technologies. SPIE Conference on Intelligent Robots and Computer Vision XIV: Algorithms, Techniques, Active Vision, and Material Handling. In David P. Casasent, Editor, Proceedings SPIE 2588. Philadelphia, Pennsylvania, 22 - 26 Oct. 1995. SPIE - The Society for Optical Engineers, Bellingham, Washington. Pp. 88 - 102.

Hasemann, J.-M. & Heikkilä T. 1993a. A new Approach towards Monitoring in Intelligent Robots. Proceedings of the Scandinavian Conference on Artificial Intelligence (SCAI'93). Stockholm, Sweden, May 4 - 7, 1993. IOS Press, Amsterdam, Netherlands. Pp. 60 - 76.

Hasemann, J.-M. & Heikkilä T. 1993b. An Embedded Monitoring System for Intelligent Robots. Proceedings of the IEEE/RSJ Intelligent Robots and Systems Conference (IROS'93). Yokohama, Japan, July 26 - 30, 1993. IEEE, Piscataway, New Jersey. Pp. 1431 - 1438.

Hasemann, J.-M. & Känsälä, K. 1994a. A Fuzzy Controller to Prevent Wheel Slippage in Heavy Duty Off Road Vehicles. Proceedings of the 44th IEEE Vehicular Technology Conference (VTC'94). Stockholm, Sweden, June 8 - 10, 1994. IEEE, Piscataway, New Jersey. Pp. 1108 - 1112.

Hasemann, J.-M. & Känsälä, K. 1994b. Avoiding Slippage with Fuzzy Logic. Industrial Horizons 1/94. VTT Communications, Espoo, Finland. Pp. 12 - 15.

Hasemann, J.-M., Känsälä, K. & Pok, Y.-M. 1994. Analysis and Design of a Fuzzy Controller to Prevent Wheel Slippage in Heavy Duty Off Road Vehicles. Proceedings of the IEEE World Congress on Computational Intelligence (FUZZ-IEEE 94). Orlando, Florida, June 26 - July 4, 1994. IEEE, Piscataway, New Jersey. Pp. 1069 - 1074.

Hecking, M. 1993. Eine logische Behandlung der verteilten und mehrstufigen Planerkennung. Reihe Informatik, Verlag Shaker, Aachen, Germany. 248 pp. (In German.)

Heikkilä, T. & Röning, J. 1992. PEM Modelling: A Framework for Designing Intelligent Robot Control. Journal of Robotics and Mechatronics, Vol.4, Number 5, pp. 432 - 444.

Hendler, J. & Subrahmanian, V. S. 1990. A Formal Model of Abstraction for Planning. University of Maryland, Computer Science Technical Report Series, CS-TR-2480. College Park, Maryland. 25 pp.

Hendler, J., Tate, A. & Drummond, M. 1990. AI Planning: Systems and Techniques. AI Magazine, Vol. 11, Number 2, Summer 1990, pp. 61 - 77. (ISSN 0738-4602)

Hexmoor, H. 1995. Smarts are in the Architecture. Proceedings of the 1995 AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents, Stanford University, March 27 - 29, 1995. AAAI/MIT Press, Boston, Massachusetts. Pp. 116 - 122. (Currently at ftp://hobbes.jsc.nasa.gov/pub/korten/ spring_symposium/submissions)

Hexmoor, H., Lammens, J. & Shapiro, S. C. 1993. Embodiment in GLAIR: A Grounded Layered Architecture with Integrated Reasoning for Autonomous

Agents. Dankel, D. (Ed.) Proceedings of the Florida AI Research Symposium: Pp. 325 - 329. (Also available as TR-93-10, Computer Science Department, SUNY at Buffalo.)

Ingrand, F. F., Chatila, R., Alami, R. & Robert, F. 1995. Embedded Control of Autonomous Robots using Procedural Reasoning. Proceedings of the International Conference on Robotics and Automation (ICRA'95), Nagoya, Japan. IEEE Computer Society Press, Los Alamitos, California. (Currently available at: ftp://ftp.laas.fr/pub/ria/felix/ icar95.ps.gz)

IS Robotics 1995. Company Profile. IS Robotics, Somerville, Massachusetts. 10 pp. (Currently at http://haifa.isx.com/~isr/)

JIRA (Japan Industrial Robot Association) 1994. The Specifications and Applications of Industrial Robots in Japan. Non-Manufacturing Fields - 1994, Tokyo, Japan. 370 pp.

Kaelbling, L. P. & Rosenschein, S. 1989. Action and Planning in Embedded Agents. Teleos Research Report TR-89-07. Teleos Researco, Menlo Park, California. 22 pp.

Känsälä, K. & Hasemann, J.-M. 1994a. Fuzzy Control against Wheel Slippage. Proceedings of the International Fuzzy Systems and Intelligent Control Conference (IFSICC94). Louisville, Kentucky, March 14 - 16, 1994. Institution for Fuzzy Systems and Intelligent Control, Inc. Louisville, Kentucky. Pp. 149 - 158.

Känsälä, K. & Hasemann, J.-M. 1994b. An Embedded Distributed Fuzzy Logic Traction Control System for Vehicles with Hydrostatic Power Transmission. Proceedings of the 7th Mediterranean Electrotechnical Conference (MELECON'94). Antalya, Turkey, April 12 - 14, 1994. IEEE, Piscataway, New Jersey. Pp. 719 - 726.

Känsälä, K. & Hasemann, J.-M. 1994c. Simple and Robust: Fuzzy for Antislip. Proceedings of the 6th IEEE Euromicro Workshop on Real-time Systems. Vaesteraas, Sweden, June 15 - 17, 1994. IEEE Computer Society Press, Los Alamitos, California. Pp. 240 - 245.

Känsälä, K. & Hasemann, J.-M. 1995. An Embedded Fuzzy Anti Slippage System for Heavy Duty Off Road Vehicles. Information Sciences, Vol. 4, pp. 1 - 27.

Khatib, O. 1986. Real-time Obstacle Avoidance for Manipulators an Mobile Robots. The International Journal of Robotics Research, Vol. 5, Nr. 1, pp. 90 - 98.

Kosko, B. 1992. Neural Networks and Fuzzy Systems: a Dynamical Systems Approach to Machine Intelligence. Prentice Hall, Englewood Cliffs. 449 pp.

Kuipers, B. 1986. Qualitative Simulation. Artificial Intelligence, Vol. 29, pp. 289 - 338.

Latombe, J. C. 1991. Robot Motion Planning. Kluver Academic Publishers, Boston, Massachusetts. 652 pp.

Matsushita, T., Sakane, S., Heikkilä, T. & Vähä, P. 1993. A Rule-Based Control Concept for Paper Roll Manipulation. Proceedings of the IEEE/RSJ Intelligent Robots and Systems Conference (IROS'93). Yokohama, Japan, July 26 - 30, 1993. IEEE, Piscataway, New Jersey. Pp. 414 - 419.

Meijer, L. O., Hertzberger, T. L., Mai, E.; Gaussens, F. & Arlabosse, ??? 1990. Exception Handling System for Autonomous Robots Based on PES. Collection Intelligent Autonomous Systems: Proceedings of an International Conference, Amsterdam, the Netherlands. Ed. By T. Kanade, F. C. A. Groen & L. O. Hertzberger. Stichting International Congress of Intelligent Autonomous Systems Amsterdam. Pp. 65 - 77. ISBN 90-800410-1-7.

Miller, D. P. 1989. Execution Monitoring for a Mobile Robot System. Proceedings of the SPIE (Society of Photo-Optical Instrumentation Engineers) Conference on Intelligent Control, Cambridge, Massachusetts. SPIE-The International Society for Optical Engineering, Bellingham, Washington. Pp. 36 - 43.

Nianzu, Z., Ruhui, Z. & Maoji, F. 1994. Fuzzy Control used in Robotic Arm Position Control. Proceedings of the IEEE World Congress on Computational Intelligence (FUZZ-IEEE 94), Orlando, Florida, June 26 - July 4, 1994. IEEE, Piscataway, New Jersey. Pp. 1484 - 1489.

Nilsson, N. 1994. Teleo-Reactive Programs for Agent Control. Journal of Artificial Intelligence Research, Vol. 1, pp. 139 - 158. (Currently available at http://ic-www.arc.nasa. gov/ic/jair-www/volume1/nilsson94a.ps)

Noreils, F. R. & Chatila, R. G. 1989. Control of Mobile Robot Actions, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-89), Scottsdale, Arizona, May 14 - 19, 1989. IEEE Computer Society Press, Los Alamitos, California. Pp. 701 - 707.

Noreils, F. R. & Prajoux R. 1991. From Planning to Execution Monitoring Control for Indoor Mobile Robot. Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-91), Sacramento, California, April 1991. IEEE Computer Society Press, Los Alamitos, California. Pp. 1510 - 1517.

Payton, D. W. 1986. An Architecture for reflexive Autonomous Vehicle Control. Proceedings 1986 IEEE International Conference on Robotics and Automation (ICRA'86), San Francisco, California, April 7 - 10. IEEE Computer Society Press, Los Alamitos, California. Pp. 1838 - 1845.

Penberthy, J. S. & Weld, D. 1992. UCPOP: A Sound, Complete, Partial-Order Planner for ADL. In: B. Nebel, C. Rich & W. Swartout (eds.) Principles of Knowledge Representation and Reasoning. Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR-92), Cambridge, Massachusetts, October 1992. Morgan Kaufmann, San Mateo, California. Pp. 103 - 114.

Puppe, F. 1990. Problemlösungsmethoden in Expertensystemen, (Studienreihe Informatik). Springer, Berlin, Heidelberg, New York. 256 pp. ISBN 3-540-53231-5. (In German.)

Reece, G. A. & Tate, A. 1994. Synthezing Protection Monitors from Causal Structure. Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS'94). Chicago, Illinois, June 13 - 15, 1994. AAAI Press, Menlo Park, California. Pp. 146 - 151. Also available as report AIAI-TR-148, Aritificical Intelligence Application Institute, University of Edinburgh, Edinburgh, UK.

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. Artificial Intelligence, Vol. 5, No. 2, pp. 115 - 135.

Saffioti, A., Konolige, K. & Ruspini, E. 1993a. A Multivalued Logic Approach to Integrating Planning and Control. Technical Note No. 533. SRI International, Menlo Park, California. 94 pp. (Also to appear in Artificial Intelligence.)

Saffioti, A., Ruspini, E. & Konolige, K. 1993b. A Fuzzy Controller for Flakey, an Autonomous Mobile Robot. Technical Note No. 529. SRI International, Menlo Park, California. 33 pp.

Schneeberger, J. 1993. Plan Generation by Linear Deduction.Tasso Report Nr. 51, Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD), Sankt Augustin, TH Darmstadt/FB Informatik Darmstadt, S.E.P.P. GmbH Röttenbach. 156 pp.

Schoppers, M. J. 1987. Universal Plans for Reactive Robots in Unpredictable Environments. Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI'87), Milan, Italy, August 23 - 28, 1987. International Joint Conference on Artificial Intelligence, Menlo Park, California. Pp. 1039 - 1046.

Schoppers, M. J. 1989. In Defense of Reaction - Plans as Caches. AI Magazine, Winter 1989, pp. 51 - 60.

Sheridan, T. B. 1992. Telerobotics, Automation, and Human Supervisory Control. MIT Press, Cambridge, Massachusetts. 393 pp.

Shoham, Y. 1992. Agent-oriented Programming. Artificial Intelligence, Vol. 60, pp. 51 - 92.

Simmons, R. 1990. Concurrent Planning and Execution for a Walking Robot. Technical Report, CMU-RI-TR-90-16. Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania. 14 pp.

Simmons, R. 1994. Structured Control for Autonomous Robots. IEEE Transactions on Robotics and Automation, Vol. 10, No. 1, February 1994. 10 pp.

SR (Suomen Robotiikkayhdistys ry), 1993. Palvelurobotiikka. Tutkimuksen loppuraportti. Suomen Robotiikkayhdistys ry (Robotics Society in Finland), Helsinki, Finland. 24 pp. (In Finnish.)

Sundermeyer, K. 1993. Modellierung von Agentensystemen. Daimler Benz Forschung Systemtechnik, Technischer Bericht F2S-93-014, Daimler Benz AG Forschung und Technik Forschung Systemtechnik, Berlin, Germany. 33 pp. (In German.)

Tate, A. 1977. Generating Project Networks. Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-77). Cambridge, Massachusetts, USA. International Joint Conference on Artificial Intelligence, Menlo Park, California. Pp. 888 - 893.

Tate, A. 1994. The Emergence of “Standard” Planning and Scheduling System Components - Open Planning and Scheduling Architectures. Proceedings of the EWSP'93 - 2nd European Workshop on Planning. Vadstena, Sweden, December 9 - 11, 1993. IOS Press, Amsterdam, Netherlands. Pp. 14 - 32.

Voudouris, C., Chernett, P., Wang, C. J. & Callaghan, V. L. 1994. Fuzzy Hierarchical Control for Autonomous Vehicles. University of Essex, Dept. of Computer Science, Technical Report CSM-216, Colchester, UK. 8 pp. (Currently available at ftp://hp1.essex.ac.uk/pub/csc/technical-reports/CSM-216.ps.Z)

Wilkins, D. E. & Myers, K. L. 1995. A Common Knowledge Representation for Plan Generation and Reactive Execution. Journal of Logic and Computation, Vol. 5, No. 6, pp. 731 - 761. (Currently at http://www.ai.sri.com/ people/wilkens/papers.html)

Wilkins, D. E. 1983. Representation in a Domain-Independent Planner. Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-83), Karlsruhe, Germany. International Joint Conference on Artificial Intelligence, Menlo Park, California. Pp. 733 - 740.

Wilkins, D. E., Myers, K. L, Lowrance, J. D. & Wesley, L. P. 1994. Planning and Reacting in Uncertain and Dynamic Environment. Journal of Experimental and Theoretical AI, Vol. 6, pp. 197 - 227. (Currently at http://www. ai.sri.com/people/wilkens/papers.html)

Yang, Q. & Tenenberg, J. D. 1990. ABTWEAK: Abstracting a Nonlinear, Least Commitment Planner. Proceedings of the AAAI-90, July 29 - August 3, 1990. AAAI Press, Menlo Park, California. Pp. 204 - 209.

Definition 1. Modal Truth Criterion (MTC). "A proposition p is necessarily true in a situation s iff two conditions hold: (1) there is a situation t equal or necessarily previous to s in which p is necessarily asserted; (2) and for every step V possibly before s and every proposition q possibly codesignating with p which V denies, there is a step E necessarily between V and s which asserts r, a proposition such that r and p codesignate whenever p and q codesignate. The criterion for possible truth is exactly analogous, with all the modalities switched (read "necessary" for "possible" and vice versa)". *(Chapman 1987)*

Definition 2. (Plan Component Class) A plan component class is a system *plan_component_class = (pc_type , pc_allowed_parameters , pc_expansions, conditions , instantiation)* :

pc_type : specifies the type of the plan component class;
$pc\_type \in \{abstract, atomic, t\text{-}virtual, i\text{-}virtual, g\text{-}virtual\}$
pc_allowed_parameters: is the set of allowed parameter vectors.
pc_expansions: is a set of plan expansions. (abstract plan components only, see later)
conditions: is a set of conditions (virtual plan components only, see later) used for monitoring.
instantiation: $plan\_component\_class * parameter\_vector \rightarrow plan\_component$
the instantiation function for the plan component class. It is defined for $parameter\_vector \in pc\_allowed\_parameters$.

Definition 3. Plan Component. A plan component is a n-tuple:
plan_component = (class , parameters , m_conds , status):

class: is the plan component class of the plan component.
parameters: is a vector of instantiation parameters of the plan component; $parameters \in pc\_allowed\_parameters$ of plan component class "*class*" .
m-conds: a set of condition - parameter_vector pairs. Used for monitoring. Resulting during instantiation by parametrisation of plan component class conditions.
$status \in \{planning, executing, executed\}$; execution, planning status.

Definition 4. Plan Component Universe. The plan component universe PC is the set of all possible plan components. PC consists of atomic plan components TPC, i.e. terminal non-decomposable plan components, and abstract plan components APC which are further decomposed. It holds:

$PC = APC \cup TPC, \text{ with } APC \cap TPC = \emptyset$
$\forall pc \in APC \Leftrightarrow c = class_{pc} \wedge pc\_type_c = \{abstract\}$
$\forall pc \in TPC \Leftrightarrow c = class_{pc} \wedge pc\_type_c = \{atomic, t\text{-}virtual, i\text{-}virtual, g\text{-}virtual\}$

Definition 5. Doubly Pointed Graphs / Plans. A double pointed graph over an alphabet *PC=APC* $\cup$ *TPC* of terminal (*TPC*; atomic/virtual plan components) and non-terminal (*APC*; abstract plan components) plan components is a system *H* = *(Nodes , Edges , label , from , to , begin, end)* with:

Nodes = a set of nodes, i.e. time points
Edges = a set of edges, i.e. plan components
label : Edges $\rightarrow$ *PC* = an edge labelling function, i.e. plan component
from : *Edges* $\rightarrow$ *Nodes* = the source node (from-node of an edge)
to : *Edges* $\rightarrow$ *Nodes* = the target node (to-node of an edge)
begin $\in$ *Nodes* = the start node of the graph
end $\in$ *Nodes* = the end node of the graph, with begin≠end
from ≠ to are such that
no loops exist
all nodes and edges are connected
$\forall edge \in Edges \;\; from(edge) \neq end$
$\forall edge \in Edges \;\; to(edge) \neq begin$
$\forall node \in Nodes \;\; v \neq begin \Rightarrow \exists \, edge \in Edges \;\; from(edge) = node$
$\forall node \in Nodes \;\; v \neq end \Rightarrow \exists \, edge \in Edges \;\; to(edge) = node$

Definition 6. Monitoring Predicates relate conditions (m_conds) as part of (virtual) plan components to each other and sensor readings in order to facilitate monitoring. *m* and *n* are propositions (conditions). *s* denotes a sensor reading.

matching(m,s) $\rightarrow$ *{True, False}*
A proposition m (of the description language) matches with given sensor readings *s*, or not.
not_matching(m,s) $\rightarrow$ *{True, False}*
A proposition m does not agree with given sensor readings s, or not.
consistent(m,n) $\rightarrow$ *{True, False}*
Two propositions are consistent, if the first one implies the second one, i.e. m $\rightarrow$ n, e.g. *consistent(x > 23, x > 12) = True*. The following does not necessarily hold: *consistent(m,n)* $\Leftrightarrow$ *consistent(n,m)*
not_consistent(m,n) $\rightarrow$ *{True, False}*
Two propositions are not consistent, if the second one does not imply the first one, i.e. $m \rightarrow \neg n$. The following does not hold:
consistent(m,n) $\Leftrightarrow$ $\neg$*not_consistent(m,n)*

Definition 7. Context-free Edge Replacement Rule. A context-free edge replacement rule (*CFERR*) over an alphabet *PC=APC* $\cup$ *TPC* of nonterminal symbols *APC* (abstract plan components) and terminal symbols *TPC* (atomic/virtual plan components) is defined as (G_{PC} is the set of all doubly pointed graphs over PC):

CFERR = (lhs, rhs), with

  $lhs \in APC$, left hand side

  $rhs \in G_{PC}$, right hand side,

Definition 8. Context-free Edge Replacement Grammar.A context-free edge replacement grammar (*CFG*) over an alphabet $PC = TPC \cup APC$ (plan components) is a system $G = (TPC, APC, P, Z)$ with

G_{PC} =	is the set of all doubly pointed graphs over *PC*;
TPC =	is the set of nonterminal symbols; (physical/virtual plan components)
APC =	is the set of terminal symbols; (atomic plan components)
P =	is a set of *CFERRs* over $PC = TPC \cup APC$ and G_{PC}
$Z \in G_{PC}$	is an initial graph; (which by application of *CFERR*s is successively decomposed creating the language of the *CFG*).

Definition 9. Plan Expansion.A plan expansion pex_x*=(peaf, pef, app, CFERR)* is associated with a plan component class x such that it extends a *CFERR* by a plan expansion applicability function, a plan expansion function, and a plan expansion preconditon predicate.

peaf: parameters $\rightarrow$ *{True , False};* the plan expansion applicability function maps a parameter vector to True or False, depending whether or not this expansion is applicable for the given parameter vector.

pef: parameters $\rightarrow$ *new_parameters***;* the plan expansion functions determines the parameter vectors for all plan components within the right hand side of CFERR.

app: aspects_of_interest $\rightarrow$ *[0...1];* the applicability function app maps the operational context, such as the current situation given by sensor readings or the world model, to a heuristic notion of applicability of the plan expansion.

CFERR: a context-free edge replacement rule such that the left hand side of this rule equals x.

Definition 10. Application of Plan Expansions. Plan component decomposition is based on plan expansions. The plan expansion with the highest applicability value *app* is chosen.

The application of the context-free Edge Replacement Rule (*CFERR*) of this plan expansion maps a Source Graph *S* onto a Target Graph *T* ($S,T \in G_{PC}$) such that a plan component *p* within *S* with a plan component class matching the *lhs* of the *CFERR* is augmented by the right hand side *rhs* of the *CFERR* which is inserted between *from(p)* and *to(p)* in *T*.

Plan components new in T are instantiated (are given parameter vectors) by applying the plan expansion function pef with the parameter vector of p.

Definition 11. Establisher and Violators. A plan is given as a set of nodes N and a set of precedence relationships *R*. A precedence relationship $(p, q) \in R$ denotes that *p* occurs prior to *q* within the plan, $p,q \in N$. *k* and *l* are propositions. $goal\_state_q$ is a description of the state changes after execution of *q*. Although this definition of plans is different than the one given earlier for plans as doubly pointed graphs, both representations are equivalent and can be mutually translated into each other without loss of generality with respect to the following definitions.

Establishers are then defined as:

$$q \in Establishers(p,k) \Leftrightarrow consistent(k,l) \wedge l \in goal\_state_q \wedge (p, q) \notin R.$$

Violators are defined as:

$$q \in Violators(p,k) \Leftrightarrow not\_consistent(k,l) \wedge l \in goal\_state_q \wedge (p, q) \notin R.$$

Definition 12. Plan Validity Monitoring (I). A plan is given as a set of nodes N and a set of precedence relationships *R*. A precedence relationship $(p, q) \in R$ denotes that *p* occurs prior to *q* within the plan, $p,q \in N$.

Possible conflicts are anticipated, if one of the following conditions hold true:

$\exists k\ \forall x \in Establishers(p,k)\ \exists y \in Violators(p,k) \Rightarrow (y,x) \notin R$, i.e.,

for all establishers there exists at least one proposition *k* of a plan component *p*, such that there exists at least one violator *y* of *k*, which is possibly executed after *x* .

$\exists k\ Establishers(p,k) = \emptyset \wedge Violators(p,k) \neq \emptyset$, i.e.,

at least one proposition *k* of a plan component *p* exists which has no establisher but at least one violator.

Pending conflicts are anticipated, if

$\exists k\ \exists y \in Violators(p,k)\ \forall x \in Establishers(p,k) \Rightarrow (x, y) \in R \wedge (y, p) \in R$,

i.e., if for at least one violator *y* of at least one proposition *k* of a plan component *p,* all establishers are planned to be executed before the violator. This condition indicates a planning error and can be removed if the planner produces only correct plans, e.g. TWEAK or ABTWEAK. However, if this is not guaranteed it is worthwhile to check this condition, too. The case that the violated condition may be reestablished by an unknown outside effect is ignored here.

$\exists k\ Establishers(p,k) \neq \emptyset \wedge (\neg matching(k,s) \vee \exists x \in Violators(p,k) \Rightarrow (x, p) \in R)$,

i.e., if for at least one proposition *k* of a plan component *p* no establisher exist and the proposition *k* does not match with the or sensor readings or at least one violator exists which is executed prior to *p.*

Definition 13. Plan Validity Monitoring (II). Plan components which may be executed concurrently have to be further checked for conflicts. A plan is given as a set of nodes N and a set of precedence relationships R. A precedence relationship $(p, q) \in R$ denotes that p occurs prior to q within the plan, $p, q \in N$.

A conflict is imminent if at least one pair of (parallel) propositions exists which is not consistent ($p, q \in P$, $(p, q) \notin R$, $(q, p) \notin R$, $status_p$ = *"planning"* , $status_q$ = *"planning"*):

$\exists k \in m\_conds_p \wedge pc\_type_p \in$ {T-virtual, I-virtual, G-virtual} $\wedge$

$\exists l \in m\_conds_q \wedge pc\_type_q \in$ {T-virtual, I-virtual, G-virtual}

$\Rightarrow$ *not_consistent*(k, l), with $pc\_type_q = pc\_type$ of class of q.