Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Michael Beetz

Concurrent Reactive Plans

Anticipating and Forestalling Execution Failures

Springer

Series Editors

Jaime G. Carbonell,Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Author

Michael Beetz
Universität Bonn
Institut für Informatik III
Römerstr. 164, 53117 Bonn, Germany
E-mail: beetz@cs.uni-bonn.de

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Beetz, Michael:
Concurrent reactive plans : anticipating and forestalling execution failures / Michael Beetz. - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1772 : Lecture notes in artificial intelligence)
ISBN 3-540-67241-9

CR Subject Classification (1991): I.2.9, I.2.11, I.2.8, I.2.6, I.2, C. 2.4, F.3, D.1.3, D.3.2

ISBN 3-540-67241-9 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10719790 06/3142 5 4 3 2 1 0

Dedicated to the memory of my mother
Renate Beetz

and my grandparents
Elise and Fritz Lehmann

Abstract

Autonomous robots that accomplish their jobs in partly unknown and changing environments often learn important information while carrying out their jobs. To be reliable and efficient, they have to act appropriately in novel situations and respond immediately to unpredicted events. They also have to reconsider their intended course of action when it is likely to have flaws. For example, whenever a robot detects another robot, it should predict that robot's effect on its plan and — if necessary — revise its plan to make it more robust. To accomplish these patterns of activity we equip robots with *structured reactive plans (SRPs)*, concurrent control programs that can not only be interpreted but also reasoned about and manipulated. These plans specify how the robot is to respond to sensory input in order to accomplish its jobs.

In this book we describe a computational model of forestalling common flaws in autonomous robot behavior. To this end, we develop a representation for SRPs in which declarative statements for goals, perceptions, and beliefs make the structure and purpose of SRPs explicit and thereby simplify and speed up reasoning about SRPs and their projections. We have also developed a notation for transforming SRPs, which does not only make the physical effects of plan execution explicit, but also the process of plan interpretation, as well as temporal, causal, and teleological relationships among plan interpretation, the world, and the physical behavior of the robot. Using this notation a planning system can diagnose and forestall common flaws in robot plans that cannot be dealt with in other planning representations. Finally, we have extended the language for writing SRPs with constructs that allow for a flexible integration of planning and execution and thereby turned it into a single high-level language that can handle both planning and execution actions.

Experiments in a simulated world show that by simultaneously forestalling flaws and executing SRPs, the robot can perform its jobs more reliably than, and almost as efficiently as, it could using fixed control programs.

Acknowledgements

This manuscript is a revised version of my dissertation written at the Artificial Intelligence Laboratory at Yale University. In working on this dissertation, I have greatly benefited from the help and support of many people. Here are those I would like to thank especially.

First and foremost, I want to thank my advisor, Drew McDermott. He provided an excellent (intellectual and software) environment in which I could pursue my research. He was always there when I needed advice and gave me the freedom to explore my ideas. During my tenure at Yale I have benefitted immensely from his broad intuitions and his ability to turn vague ideas into working LISP programs.

Ken Yip has been a good friend and mentor throughout. In particular, I want to thank him for teaching me that "knowing" the solution before understanding the problem is not a good way to start one's research. I hope, I won't forget too often. Greg Hager and Reid Simmons were the other two members of my Dissertation committee. Their careful reading, critique, and insightful comments have greatly improved this dissertation. I would also like to thank Pam Sturner and Zach Dodds for taking the time to read and comment on an earlier draft of this dissertation. My gratitude can best be measured by the length of this document. Sean Engelson deserves credit for being a good friend, perfect office mate, and discussion partner.

Since the completion of my thesis I have worked as a member of the RHINO team at the University of Bonn and applied techniques described in this book to problems in autonomous robot control. I would like to thank Prof. Armin B. Cremers for his support and for providing such an excellent research environment. Carrying out successful research in autonomous robot control is not possible without being a member of a team. I had the luck of joining one of the best teams: the RHINO team. I would like to thank especially the following members and alumni: Tom Arbuckle, Thorsten Belker, Wolfram Burgard, Dieter Fox, Henrik Grosskreutz, Dirk Hähnel, Dirk Schulz, Gerhard Lakemeyer, and Sebastian Thrun.

Most importantly, I want to thank my wife Annette and children Nicola and Fabian for their love and support and keeping up with me writing this manuscript.

Table of Contents

List of Figures

1. Introduction

Ja mach nur einen Plan
Sei nur ein gro es Licht!
Und mach dann noch nen zweiten Plan
Gehn tun sie beide nicht.
Bertolt Brecht

Lied von der Unzulänglichkeit menschlichen Strebens[1],
Dreigroschenoper.

Compared to current autonomous robots, people get around in their environments easily and do their everyday activities reliably and e ciently. People accomplish complex jobs in changing and partly unknown environments; they manage several jobs at a time, resolve conflicts between interfering jobs, and act appropriately in unexpected and novel situations. They even reconsider their course of action in the light of new information. Truly autonomous service robots that are to operate in cooperative and not speci cally engineered environments have to exhibit similar patterns of activity.

The work of an o ce courier illustrates well why human everyday activity is so e cient and reliable. An important reason is the flexible management of multiple, diverse jobs. An o ce courier usually has several deliveries and other errands to perform. He is asked to pick up packages and letters, go to particular o ces or mail boxes, and hand over or deposit packages at the right places. At any time, secretaries, o ce workers, and managers can give him new errands or cancel and revise old ones. In addition, he has regular daily and weekly errands, like cleaning his o ce.

Since an o ce courier does his daily activities over and over again and is confronted with the same kinds of situations many times, he learns [156] and uses [1] reliable and e cient routines for carrying out his daily jobs. Performing his daily activities usually does not require a lot of thought because routines are general. This generality is necessary since jobs come in many variations and require the courier to cope with whatever situations he encounters. The regularity of everyday activity also requires the courier to

[1] *Go make yourself a plan; And be a shining light! Then make yourself a second plan; For neither will come right.* The Song of Futility of all Human Endeavor, Bertolt Brecht, The Threepenny Opera.

handle interruptions that occur while carrying out his routines. Routines, like cleaning an o ce, are often interrupted if an urgent delivery is to be made, and continued later on. Therefore, the courier has to remember where he put things in order to nd them again later.

A main factor that contributes to the e ciency of everyday activity is the flexibility with which di erent routines can be interleaved. The courier often works on more than one job at a time (he checks whether a particular letter is ready when passing the o ce while doing other errands). Or, he does jobs together (if he has several deliveries for the fth floor, he makes the rounds rst to collect these deliveries, then goes to the fth floor, and nally, distributes them).

Still, everyday activity could not be as robust and e cient as it is if it consisted just of the mechanical execution of standard routines. Often the o ce courier runs into situations where his standard routines do not work, as when parts of the o ce building are cleaned, secretaries have lunch hours, meetings are held, or other jobs interfere. These events often have hidden and subtle e ects on the courier's immediate jobs. The experienced courier, unlike the less competent one, can take these pieces of information into account: he can detect whether di erent errands might interfere and forestall related problems. He can also exploit the information he gains to detect and avoid problems and take advantage of the opportunities he discovers.

To summarize, people can accomplish their daily jobs well even in situations they have not encountered yet for two reasons: they have general routines that work well in standard situations and they are, at the same time, able to recognize non-standard situations and adapt their routines to the speci c situations they encounter. Making the appropriate adaptations often requires people to predict how their routines would work in non-standard situations and why they might fail.

An important goal in the design and implementation of autonomous robot controllers is to understand and build controllers that can carry out daily jobs in o ces and factories with a reliability and e ciency comparable to people. In this book we will approach this goal by developing a computational model for robot controllers that can produce some aspects of human everyday activity and thereby achieve a higher degree of reliability and e ciency. Of course, autonomous robot controllers already exist (although only for carefully engineered environments [136] or for a few xed and speci c tasks [34]). It is also not a new idea that robot controllers equipped with AI planning techniques can be more e ective than xed robot controllers (it has been implemented in order to control several robots [136, 164, 79]). The main characteristic of such deliberative robot controllers is that they carefully reason through all actions before they execute them. Deliberative robot controllers sense their environments in order to maintain a symbolic representation of the relevant aspects of their environments. Based on this symbolic world model deliberative agent controllers construct optimal courses of action that can provably achieve the given jobs. It has, however, turned out that the conditions under which provably correct courses of actions can be constructed automatically cannot be satis ed by autonomous robots that are to act in human environments, like o ces. Such robots act in partly unknown and changing worlds, have imperfect control over their e ectors, and have only limited computational resources. Where the assumed conditions do not hold, robot controllers that compute optimal sequences of actions and execute them afterwards are only of limited use [37].

1.1 The Approach

In this book we investigate how to accomplish reliability and e ciency through *structured reactive controllers*, collections of concurrent control routines that specify routine activities and can adapt themselves to non-standard situations by means of planning. We call the controllers reactive because they can respond immediately to asynchronous events and manage concurrent control processes. We call them structured because they use expressive control abstractions to structure complex activities. Structured reactive controllers execute three kinds of control processes: routine activities that handle standard jobs in standard situations, monitoring processes that detect non-standard situations, and planning processes that adapt, if necessary, routine activities to non-standard situations (see gure 1.1).

Structured reactive controllers work as follows. When given a set of jobs, the structured reactive controller retrieves default routines for individual jobs and executes the routines concurrently. These routine activities are general and flexible they work for standard situations and when executed con-

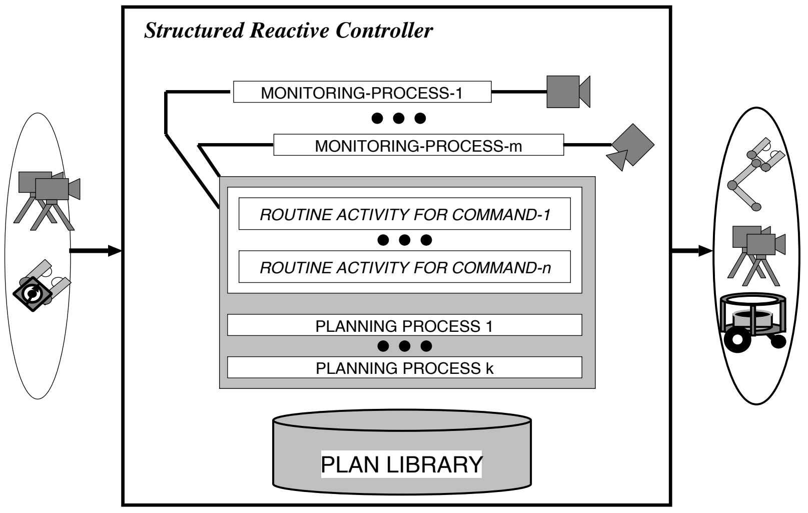

Structured Reactive Controller
MONITORING-PROCESS-1
MONITORING-PROCESS-m
ROUTINE ACTIVITY FOR COMMAND-1
ROUTINE ACTIVITY FOR COMMAND-n
PLANNING PROCESS 1
PLANNING PROCESS k
PLAN LIBRARY

Fig. 1.1. Top-level view of structured reactive controllers.

currently with other routine activities. Routine activities can cope well with partly unknown and changing environments, run concurrently, handle interrupts, and control robots without assistance over extended periods. For standard situations, the execution of these routine activities causes the robot to exhibit an appropriate behavior in achieving their purpose. While it executes routine activities, the robot controller also tries to determine whether its routines might interfere with each other and watches out for non-standard situations. If it encounters a non-standard situation it will try to anticipate and forestall behavior flaws by predicting how its routine activities might work in the non-standard situation and, if necessary, revising its routines to make them robust for this kind of situation. Finally, it integrates the proposed revisions smoothly into its ongoing course of actions.[2]

Structured reactive controllers have the competence to carry out behavior speci cations such as the following ones:

perform jobs concurrently, but if necessary perform them in an order in which they do not interfere;
pick up an empty box, whenever you see one but only if it helps you to perform your jobs more e ciently; and
protect a goal only if you expect that another robot might destroy it.

These three behavior speci cations are characterized by the following properties: they notice non-standard situations, the decisions underlying the behavior require foresight, and they change the future course of action.

On the surface, it seems obvious that adding the capability of making decisions that require foresight will enable robots to perform their jobs better. However, in a more careful analysis we will nd that the statement is not trivial for two reasons. First, it is not clear that making such decisions is feasible: the decisions have to be made in limited time and based on sparse and sometimes faulty information provided by the robot's sensors. In addition, the implementation of these decisions requires structured reactive controllers to make subtle revisions in complex and flexible robot control programs. Second, while the robot is making the lengthy strategic decisions it keeps executing its controller and therefore has to smoothly integrate the results of its deliberations into its ongoing activities.

The research described in this book makes two claims. First, the process of anticipating and forestalling flaws in the behavior of a robot can be successfully realized if we implement robot controllers as *structured reactive controllers* and realize planning e orts as computational processes that are part of routine activities. Second, robot controllers that anticipate and forestall large varieties of behavior flaws can outperform robots with xed control-

[2] This book does not address the problems of learning and remembering the revisions and the situations in which they are appropriate. Chapter 8.2.2 lists components of structured reactive controllers that can be constructed, maintained, and improved using automated learning techniques.

lers if the robots are to perform varying everyday activities in changing and partly unknown environments.

The thesis is supported by developing a computational model for the process of anticipating and forestalling behavior flaws; demonstrating how this model can be incorporated into a robot controller; and performing experiments that show that a simulated robot with a controller implemented in this way outperforms the same robots with xed robot controllers.

I should at this point state what I mean when I say one robot controller is better than another one. A lot of planning research is concentrating on properties like the correctness and optimality of robot plans and the complexity and completeness of planning algorithms. However, even *if* we had planning algorithms that could compute correct and optimal plans in polynomial time[3] this would not imply that these algorithms are actually capable of improving a robot's problem-solving behavior. Even polynomial-time algorithms might not be fast enough, and correctness and optimality of plans can only be guaranteed with respect to the world model used in their generation, and not with respect to the actual state of the world. Disembodied properties of planning algorithms like the correctness and optimality of plans, and the algorithms' complexity are neither necessary nor su cient criteria for their usefulness.

An important measure for the quality of an embedded planning algorithm is how much the planning algorithm improves the performance of the robot's behavior. We would like the robot to perform as follows: if jobs can be appropriately performed by routine activities an observer should not notice any planning. In cases where the routine activity performs poorly the planning robot should still, with a high probability, exhibit an appropriate behavior. Why probably? Why appropriate rather than optimal? At best the planning e orts can with a high probability achieve a better behavior: relevant information might not always be available and available information might not always be correct. And even if the planning e orts are computing an optimal plan, the resulting behavior of the robot still might not be optimal because planning takes time and the world does not stand still. When installing a revision, e orts that have already been spent on executing the routine controller might turn out to be wasted, or even worse, have to be reversed. In this book we use the following interpretation of better: an embedded planning algorithm can be said to improve the performance of a robot if, for a representative random set of jobs, the average score obtained by the robot employing the planning algorithm is higher than the score of any xed robot controller.[4] The score is based on the number of jobs the robot has performed, the degree to which it has accomplished them, and the time it has spent.

[3] see [66, 43], for some complexity results for propositional planning.

[4] see [166] for an discussion of the utility of planning.

1.2 Technical Challenges

The implementation of structured reactive controllers presents several difficult technical challenges. The first is that causing the kind of behavior that people exhibit in their everyday activities requires sophisticated synchronization of concurrent control processes, monitoring of the execution, adequate reaction to unforeseen events, and recovery from local execution failures. Unfortunately, plans that are partially ordered sets of actions cannot describe behaviors as complex and flexible as routine activities concisely, because they lack suitable control abstractions. Therefore, we use RPL (Reactive Plan Language [124]), a structured plan language, which provides control abstractions that help to structure the complex everyday activities and make them reactive and adaptive. The concepts and control structures provided by RPL include conditionals, loops, program variables, processes, and subroutines. In addition, RPL offers high-level constructs (such as interrupts and monitors) to synchronize parallel actions and make plans reactive and robust. The downside of choosing such an expressive plan language for the implementation of routine behaviors is that the automatic inference of the structure of control routines, the purpose of subroutines, and installation of plan revisions become much harder. However, these difficulties are not caused by the expressiveness of the plan language but rather by the complexity and flexibility of behaviors the planning systems should reason about. After all, control abstractions of structured programming languages make programs more intelligible and easier to change. Therefore, when the robot plans its activities, it must not abstract away from these concepts and control structures but reason through these concepts to find causal and teleological explanations for faulty behaviors. The robot also has to use the control structures to revise its course of action.

The second challenge is the spectrum of behavior flaws that planning processes have to anticipate and forestall. Many different kinds of failures occur when autonomous robots carry out their everyday activities. These failures may be caused by imperfect sensors, imperfect control, an inaccurate and incomplete world model, and interference from other tasks. The more failures the planning processes can handle, the more robust the robot controller will be. To achieve this coverage of failures, planning processes have to distinguish among different kinds of failures. If a sensor program overlooks objects, it should be revised in a different way than if it simply confuses the shapes of objects. The plan revisions themselves require the planner to localize subroutines that should be revised — those that possibly cause or permit plan failures. If an object is overlooked in an execution scenario the planner has to revise the corresponding subplan for sensing objects. Making robot plans more robust against different failure types also requires specialized and complex plan revision algorithms; a plan revision algorithm that eliminates task interference does not have much in common with one that tries to make sensing plans more robust, or with another one that makes plans more robust against exogenous events. It may seem as if there should be a more elegant and concise theory of plan debugging as, for instance, proposed by Simmons [149], which does not require us to distinguish between so many di erent kinds of plan failures. However, in order to make theories for plan debugging concise we have to make as in classical planning strong assumptions. Simmons [149], for instance, considers only plans that are partially ordered sets of plan steps, and plan failures are combinations and instances of a small set of basic plan failures. These theories do not work for plans that try to recover from local execution failures and contain concurrent interfering control processes.

A third challenge is the tight interaction between planning and execution that is necessary to adapt parts of the robot controller as the robot learns more about its environment and to smoothly integrate revisions into partially executed robot controllers. Typically, autonomous robots that perform their jobs in partly unknown and changing environments acquire important and previously missing information during plan execution and require, therefore, a strong interaction between planning and execution: a robot perceiving an empty box should think about whether that box could help it to perform its current tasks more e ciently. If the box will be helpful, the robot then needs to plan how to use it. Or, for instance, if the robot sees another robot, it should reason about whether the other robot might interfere with some of its plans and revise these plans if necessary. Or, if a robot is to deliver a number of objects in a series of errands, it should postpone planning the errands until the exploration for the objects is completed. In all these examples the robot has to react to risks and opportunities by (re)planning subtasks and postponing planning as long as necessary information is missing. The robot also should stop planning if the information used for planning is detected to be outdated, or install plan revisions without causing problems for the robot's ongoing activities.

To implement behavior speci cations such as perform the jobs concurrently, but if necessary perform them in an order in which they do not interfere, we equip planning processes with plan revision rules such as if a goal might be clobbered by an exogenous event then stabilize the goal state immediately after achieving it. Applying the plan revision rule competently requires foresight and the revision of the intended course of action. The revision rule is general and abstract plan revision methods: it states how to revise a plan of action in terms of goals, causing and achieving subplans. Writing plan revision methods in causal and teleological terms rather than in syntactic patterns is necessary because at a syntactic level there are just too many plan revision methods necessary. Of course, stating plan revision methods in more abstract terms puts the burden on the planning processes to understand enough about a robot controller to decide whether a plan revision method applies. For instance, checking the applicability of these revision rules requires planning processes to infer whether particular parts of the plan were executed and why. Thus, the conditions in revision rules listed above refer not only to the physical e ects of plan execution, but also to the process of plan interpretation, as well as to temporal, causal, and teleological relationships among plan interpretation, the world, and the physical behavior of the robot.

1.3 Introductory Example

In this book we apply our planning techniques to errand planning in a simulated world of discrete locations. Boxes, blocks, and balls of di erent colors, textures, and sizes are spread out in the world and can move, appear, disappear, or change their appearance due to exogenous events or when manipulated by other robots. The robot controllers we develop will control a robot with two hands and two cameras. By using a camera the robot can look for objects, track objects in the camera view, and examine a tracked object. Sensor programs look for objects matching a given description and return a designator describing each matching object and its location. Since the robot's cameras are imperfect, designators will be inaccurate and the robot may overlook or hallucinate objects. To change its environment, the robot can reach out a hand to the positions of tracked objects, grasp the objects, and pick them up; however, while being grasped or carried, some objects may slip out of the robot's hands or remain stuck when the robot opens a hand.

While acting in its environment, the robot stores and updates designators and uses them to nd and manipulate the objects they describe. The known designators constitute the robot's model of the current state of its environment. This model is incomplete because it only contains designators for objects already perceived. It may even be wrong since (a) the designators are produced by noisy sensors and (b) exogenous events might have changed the world without the robot noticing. Because of these de ciencies robot controllers cannot commit to a particular course of action but rather have to specify how the robot is to react to sensory input in order to perform its job. To get an unknown red object from another location, the robot has to get there and nd a red object before it can decide how to grasp it; it also has to watch out for exogenous events and react appropriately so they do not interfere with its mission.

Suppose the world is in the state described by gure 1.2 (left) and the robot is given two commands: rst, to get all balls from location $\langle 5,5 \rangle$ to location $\langle 2,5 \rangle$ and second, to get all balls from location $\langle 2,5 \rangle$ to location $\langle 2,7 \rangle$. Thus, after the robot has performed its job the three grey balls should be at $\langle 2,7 \rangle$ and the ve white balls at $\langle 2,5 \rangle$. The situation is complicated by another robot that is cleaning at location $\langle 2,7 \rangle$.

The robot has a routine activity for achieving a given state for each object that satis es a given description. In our case this routine is instantiated to get objects that look like balls from a particular location to a speci ed destination. The routine delivers balls one by one until all balls are delivered and uses a subroutine for the delivery of a single ball. This subroutine asks

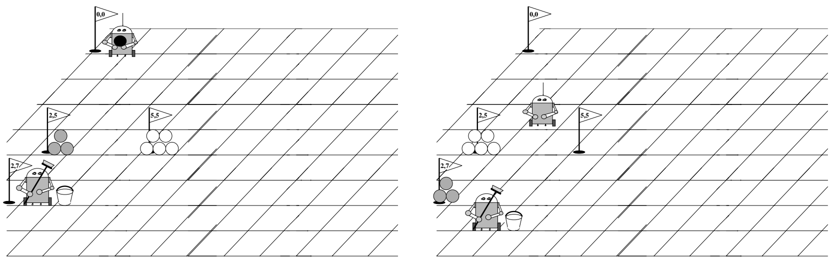


Fig. 1.2. A job given to the robot described in terms of the initial situation (left) and the goal description (right).

the robot to go to the location where the balls are supposed to be, look for a ball, and deliver it to the speci ed destination.

The routine for performing the two given user commands consists of two delivery loops that are run concurrently. Thus, when executing the routine activity for the two top-level commands the robot controller randomly activates single delivery routines from both delivery jobs until both delivery jobs are completed. When the robot starts executing the routine, it randomly picks a delivery job to work on. Suppose the rst job is the larger of the two, and the robot starts to go to location ⟨*5,5*⟩. As soon as the robot executes the routine it also activates a planning process that looks for flaws in the routine activity and tries to eliminate them. The planning process projects several random execution scenarios for the current controller. When examining the resulting execution scenarios the planner notices that balls that are supposed to be at ⟨*2,5*⟩ often end up at ⟨*2,7*⟩. Note, the delivery routine looks for any ball-like object at ⟨*2,5*⟩, no matter what the color of the ball is. On diagnosing the reason for these incorrectly performed jobs it turns out that these balls are mistakenly delivered to ⟨*2,7*⟩ by the other delivery routine after the rst one has already brought them to their destination. Our planner produces two explanations for this flaw in its behavior. One explanation is that the robot is perceptually confused: the description ball is not su cient to discriminate between the objects it is supposed to deliver from those already delivered. The second explanation is that the two delivery loops interfered with each other: a subactivity of one delivery loop clobbers a goal that has already been achieved by another delivery loop. Other possible explanations such as the robot mistakenly traveling to location ⟨*2,7*⟩ or another robot moving the balls are inconsistent with the projected execution scenarios.

Among others, our planner has the following two revision methods for failures caused by perceptual confusion:

1. **IF** the robot might become perceptually confused while performing a routine r_1 and the confusion is caused by another routine r_2 **THEN** revise the robot controller such that routine r_1 is necessarily performed before r_2.

2. **IF** the robot might become perceptually confused while performing a routine r_1 and the confusion is caused by another routine r_2 **THEN** (1) revise the routine r_2 such that it makes sure that the objects can be discriminated and (2) revise the routine r_1 such that it uses the discriminating property.

The following two are revision methods that are stored under the failure type, known as subplan interference:

1. **IF** a goal might be clobbered by another routine **THEN** make sure that the clobbering routine is executed necessarily before the clobbered goal is achieved.
2. **IF** a goal might be clobbered by another routine **THEN** revise the routine such that it reachieves the goal right before the top-level plan is completed.

These four revision methods propose four new candidate plans that are supposed to be more robust against the kind of failures that have been predicted for the routine activity. The planning process rates these four candidate plans by projecting them and estimating their utility based on the projected execution scenarios. As a result, the plans that have the additional ordering between the interfering subactivities are estimated to be better than the other candidates because they are faster. They are also considered to be better than the routine activity because all balls end up at their destinations.

As soon as the planner nds these better plans, it installs the revision corresponding to the best plan into the current robot controller. This causes the robot in our example to exhibit the following behavior. As the robot goes to $\langle 5,5 \rangle$ it cancels its current delivery and starts executing the revised controller. It therefore now goes rst to $\langle 2,5 \rangle$ to start with the other delivery job.

When it arrives at $\langle 2,5 \rangle$ it senses the other robot at $\langle 2,7 \rangle$. Robots always communicate their current jobs to each other when they meet. This way the delivery robot learns that the other robot is cleaning $\langle 2,7 \rangle$. This causes our robot to start thinking about how the cleaning robot might interfere with the delivery jobs, because other robots at places close to where our robot has to work are considered to be non-standard situations. The probabilistic model of the cleaning process used by our robot speci es that the cleaning robot might remove objects from the location it is cleaning. Thus in some of the execution scenarios projected by the planning process, balls that have been delivered are removed later by the cleaning robot. Our planner diagnoses a failure of the type goal clobbered by other robot and suggests several revisions. One suggests delivering the balls after the cleaning robot is done; another one is to stabilize the goal after it has been achieved, i.e., make sure that the balls do not disappear from their destination. Since it has no other jobs that are ready for execution, it considers the stabilizing revision to be the best transformation and revises the routine for single deliveries such that the robot prevents the balls from disappearing by nailing them down immediately after they are delivered.

1.4 Motivation

Structured reactive controllers are relevant for robotics research because so far robot controllers cannot perform the diversity of jobs and handle the variety of situations that occur in cooperative working environments. The work is also important for the eld of AI planning because this level of performance cannot be achieved without planning. Finally, the research presents novel and more powerful plan representation and inference techniques for planning research.

1.4.1 Relevance for Autonomous Robot Control

Accomplishing everyday activities in our world simulator poses problems for a simulated robot that resembles those addressed by current robotic research [54, 105, 153]. In some respects our world simulator challenges robot controllers in ways that real environments challenge real robots: the simulated environment is complex, it changes, and time in the simulator passes at a realistic speed. The sensing capabilities of the simulated robot are limited and noisy and the control over its e ectors is imperfect. As a result of these limitations the environment is partly unknown to the robot and the available information not always reliable. Because of these realistic features many behavior flaws that are exhibited by real robots also occur in the behavior of our simulated robot. The flaws include failure to accomplish jobs because of imperfect sensors, imperfect control, an inaccurate and incomplete world model, and interference from other tasks. Because of these di erent kinds of failures and because the simulator generates asynchronous events, the robot controllers have to contain complex control structures that allow them to react to asynchronous events and are able to recover from local execution failures.

In spite of such realism the world and the robot presented by the world simulator are much simpler than their real counterparts [157]. (Our world simulator is described and justi ed in chapter 2.1.) This simplicity allows us to investigate issues in intelligent robot control that cannot yet be explored using real robots. We exploit the simplicity of the world and the simulated robot in order to achieve a wider spectrum of jobs in more diverse situations.

Structured reactive controllers contribute important techniques to autonomous robot control: techniques to

1. make decisions that require foresight during plan execution;
2. forestall flaws in the behavior of autonomous on the basis of sparse information about the current state of the world; and

3. make competent decisions for dealing with ambiguous object descriptions; interferences that might be caused by other robots, and other challenges.

1.4.2 Relevance for AI Planning

Before developing a robot controller for an autonomous robot the designer has to decide whether the controller should plan. Applying AI planning techniques is expensive both in terms of development costs and allotting computational resources. Therefore, it is important to verify that the gains achieved by planning e orts will exceed the costs of applying them. If a robot were only supposed to stack blocks in the blocks world such planning capabilities could hardly be justi ed. A very simple variation of a procedure that builds towers from the bottom up is su cient to achieve all possible goal states and can almost always generate optimal sequences of actions. Such a procedure is easy to implement and very e cient computationally.

Thus, for our application we must consider whether we would be better o by designing more competent routine plans instead of expending so much e ort to implement planning capabilities. However, trying to develop more and more competent routine plans is not advantageous because it imposes a trade-o between robustness and e ciency in routine activities: robot controllers that work for all possible situations have to act as if everything might go wrong. Such paranoid behaviors are in most cases neither e cient nor appropriate. There is also a trade-o between tailoring routine plans to more speci c situations and making them more generally applicable. An advantage of making routine activities general is that the more general the routines are, the fewer are needed. A second advantage is that more general routines are more likely to be robust in situations the programmer did not envision. More special-purpose routines, on the other hand, can exploit more of the structure of the kinds of jobs they are made for and therefore be more e ective and e cient. The structured reactive approach supports the implementation of fairly general routine activities without requiring these routines to work for all situations. This is possible because the structured reactive controllers can adapt to speci c situations by automatically revising their routines.

For di erent reasons, universally robust routines are also impossible. First, designers of routine controllers usually cannot think up all the situations a robot may encounter when executing a routine. Second, the robot controller has to achieve maximal robustness for sets of routine activities with limited resources. Increasing the robustness of an activity often requires giving the activity frequent or even permanent access to particular sensors or e ectors which are then, for those stretches of time, blocked for concurrent activities of the robot. Third, it is often impossible to decide which of two routine activities is more robust.

Consider, for instance, two routines for delivering objects that can be instantiated in various ways. The rst routine simply picks up the object, goes to the destination, and puts the object down. The second is more cautious;

it tries to keep the object in the hand while going to the object's destination. Thus this routine monitors the state object in hand and reachieves this state whenever it ceases to hold. At a rst glance the second routine seems to be more robust than the rst one. However, the second routine might have flaws of its own. If the robot has no handforce sensor it might have to look with its camera whether the object is still in the hand. Thus the robot could overlook cases in which the object slips out of the robot's hand, or signal that it has lost the object even if it has not; the control program for picking up the object again might fail; or the monitoring process could cause other routines to fail by blocking them from using the camera.

To recapitulate, it is not feasible to make routines robust for all possible situations because they are to be executed in situations that have very different features. Neither is it possible to determine which of the two methods is better suited as a routine activity without knowing the distributions of commands and contexts in which the routine is to be executed. A promising alternative for many robots that have jobs that can be performed without making many decisions that require foresight is to equip the robot with routines that can handle standard situations appropriately and adapt them to the remaining situations when necessary by anticipating and forestalling behavior flaws the routines might cause in these situations.

1.5 The Computational Problem and Its Solution

This section addresses two questions: (1) how to formulate anticipating and forestalling behavior flaws in structured reactive plans as a solvable computational problem and (2) how to exploit the structure of the problem.

1.5.1 The Computational Problem

The computational problem of anticipating and forestalling behavior flaws can be stated as follows: given a robot controller in the form of a complex routine activity, a set of jobs the controller should perform, and a set of data structures describing the observations of the robot, generate alternative robot controllers that have a higher expected utility for the expected situations. The longer the planning process runs, the higher the expected utility of the robot controllers it proposes should be.

When stating computational problems for robot control we have to think about the computational resources required to solve them. Most planning systems to date use very simple models of activity and reason through plans that are partly (or totally) ordered sets of atomic actions. The planning systems usually solve the following problem: given a description of the current world state and a set of goals, determine a partially ordered set of actions such that each sequence of steps satisfying the partial order achieves a state that satisfies the goal, when executed in a state satisfying the initial world description. But even if we restrict ourselves to simple plans and describe the behavior of control routines in a very simple propositional language, the planning problem quickly becomes intractable [43]. Evaluating existing partial-order planning systems shows the effects of this intractability for typical planning problems. These planners often have to explore hundreds of partial plans in order to find plans that only require a handful of plan steps (e.g., [129]). Worse still, problems with linear complexity (as, for instance, graph searches) can become intractable by simply being restated as general planning problems [129].

One of the biggest hurdles in the realization of planning techniques for improving robot behavior is overcoming the computational complexity of AI planning methods. To reduce the computational complexity we simplify the planning problem in various ways. First, instead of trying to compute provably correct plans we content ourselves with planning techniques that, with high probability, find approximately competent plans. A second method for making the planning problem computationally easier is to devise methods for the construction of small or specialized planning problems from an existing plan and from methods for integrating the corresponding solutions into the current plan. Finally, we can restrict the set of plans we reason through, i.e., the plan language, such that planning for the restricted language is simpler.

1.5.2 The Computational Model (Data Structures & Algorithms)

Another way to make robot planning more tractable is to choose a computational model that can exploit the properties of everyday activity and the structure of the process of anticipating and forestalling behavior flaws. An important property of everyday activity we can exploit is our knowledge of routines that work well for standard situations and when run concurrently with other routines. Therefore, a robot controller can put together executable plans that have a good chance of accomplishing their jobs by retrieving the appropriate routine plans from a plan library. We also have expectations about where plans might fail and these expectations can be encoded into good and efficient methods for detecting and diagnosing flaws in predicted robot behavior. To exploit these features we base our computational model on the following principles:

1. For a given set of user commands a robot can generate a routine plan with decent expected utility by retrieving canned routine plans from a plan library and pasting them together.
2. A robot plans to avoid behavior flaws that become predictable as the robot obtains new information about its environment. The robot analyzes the reasons for predicted flaws in its behavior in order to find the most appropriate revisions.

In order to exploit the structure of everyday activity we employ a *transformational planning* technique that generates approximately competent plans and then revises these plans in order to eliminate their flaws. Despite having the same worst-case complexity as planning from rst principles [133], transformational planning is much more attractive for robot planning for several reasons. For one reason, a transformational planner can generate executable (although not necessarily successful) plans very fast. In addition, diagnosing the causes of the flaws provides valuable information to guide the heuristic search for better plans.[5] The average complexity of transformational planning can also be reduced by applying e cient heuristic plan revision methods and testing the resulting plans [134, 109] instead of relying on revision methods that guarantee improvements of the plan [41, 28]. This technique often works because checking the correctness of a solution is computationally less complex than generating correct solutions [104].

1.6 Contributions

To put this book into context we have to review briefly other approaches to autonomous robot control. We have already discussed the classical/deliberative approach in which the planner computes optimal courses of actions described in terms of plan steps. This approach requires correct and complete world models and has a tendency to cause relatively simple problems to become computationally intractable.

To avoid the problems of classical controllers in coping with real environments in real-time, several researchers developed a di erent approach, often called situated control [37, 2, 143]. In the situated control paradigm controllers are organized into interacting concurrent control processes. These processes continually sense their environment and map the resulting sensor data into e ector commands without deliberation and without maintaining large data structures. The basic working principle of situated controllers can be described as a sense-directed di erence reduction: the controller selects the appropriate e ector commands based on the incoming sensor data. An advantage of situated controllers is that they are robust against asynchronous events and sensor noise because they sense their environment at a high frequency. Situated approaches are appropriate for robots that have a small set of xed goals. Selecting commands solely on the basis of current sensor data and program state may lead to behavior that seems locally appropriate but is, from a more global point of view, a dead end for the achievement of the goals.

The hybrid layered approaches to robot control try to get the best of both worlds by combining classical planning with situated control. They assume

[5] Sussman [155]: Has it ever occurred to you that bugs are manifestations of powerful strategies of creative thinking. That, perhaps, creating and removing bugs are necessary steps in the normal process of solving a problem?

	Planning	*Execution*
Layered, Hybrid Controller	*Add, Delete Steps Add, Delete Orderings*	*Steps are Behavior Modules*
Behavior-based Controller	✗	*Concurrent Processes Reactions*

Fig. 1.3. Comparison between the hybrid, layered and situated robot control approach with respect to their planning and execution capabilities.

that all activities can, at some level of abstraction, be described as an ordered set of steps, which themselves are implemented according to the situated control paradigm. In other words, the planner con gures the situated controller for di erent problem-solving steps, thereby making the situated systems more flexible and allowing the robot to accomplish more diverse and possibly interfering goals.

If we contrast hybrid layered and situated robot control with respect to their planning capabilities and expressiveness in specifying robot behavior, we can see that hybrid layered robot controllers can plan aspects of the robot's behavior at the cost of restricting themselves to describing the robot's behavior as a set of plan steps. This restriction is made because we know how to plan with sets of action steps but not how to plan more general robot control programs. Unfortunately, this restriction also reduces the utility of planning for robot control. The purpose of the planner is to make the behavior more e cient and robust and the purpose of the control structures o ered by situated robot control languages is to make the robot's behavior robust for challenging environments. Of course, a planner cannot make the behavior of the robot more robust against many types of plan failures in challenging environments without reasoning and making use of the control structures provided by situated control languages. We are not arguing that plan steps are inappropriate abstractions. On the contrary, as you will see we will often talk about plan steps and order them. What we are saying is that plan steps and ordering constraints alone are not su cient to describe the kinds of behavior we are interested in.

The alternative is to plan the situated control of the robot itself. As we will see, by exploiting the additional information contained in situated control programs we can characterize important categories of behavior flaws that cannot be captured by sets of plan steps. Also, by making use of the control structures provided, we can make plans more robust in ways that traditional

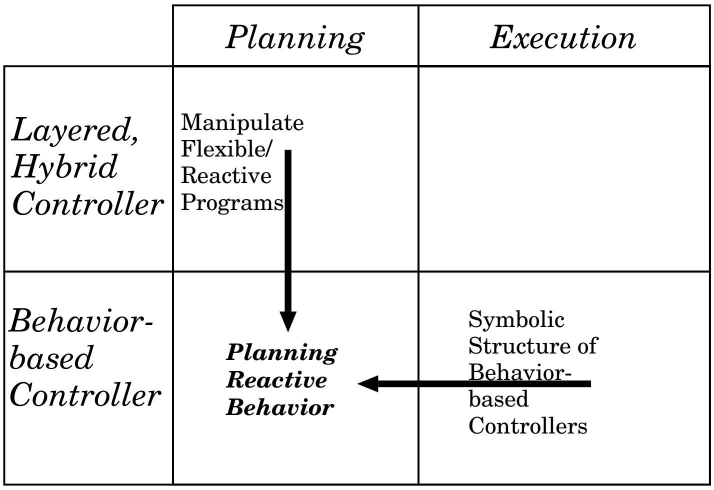

Planning
Execution
Layered, Hybrid Controller
Manipulate Flexible/ Reactive Programs
Behavior-based Controller
Planning Reactive Behavior
Symbolic Structure of Behavior-based Controllers

Fig. 1.4. Research directions in planning situated robot control.

planners cannot. Planning the situated control of a robot is the goal of a research e ort called transformational planning of reactive behavior, of which this book is part [123, 127].

Developing techniques for planning the situated control of a robot requires e orts in two areas (see gure 1.4): rst, in the area of situated control, where we have to develop symbolic representations for situated controllers that can be used by the planner to reason about and transform plans. However, it is not enough for these representations merely to allow for reasoning and transformation; rather, they have to enable the planner to use fast algorithms to carry out these planning operations. The second area in which new techniques have to be developed is in that of planning operations. We need batteries of new complex planning operations for failures, like overlooking a particular object or failures resulting from ambiguous object descriptions, that have not been studied yet. Testing whether the flawed behavior of a robot is the result of one of these events is di cult. So is revising a situated controller to make it robust against failures like these, a task which requires the plan revisions to analyze the structure of situated controllers and the purpose of routines. This book makes contributions toward both goals.

The main result of the research described in this book is the implementation of a computational model for planning that can improve the problem-solving behavior of a robot while the robot is performing its jobs. This result is important because the model can help robots that have imperfect and limited sensors, imperfect control over its e ectors, and an inaccurate and highly incomplete world model characteristics shared by many real robots. The implemented model of planning is able to anticipate and forestall behavior flaws like overlooked objects and incorrect belief that are common to many autonomous robot applications.

As has been stressed earlier, performing these revisions requires the planning processes to reason about structured reactive plans, a very expressive plan notation. One of the consequences of this choice of plan representation is that many reasoning tasks and other operations become unsolvable or highly intractable. Possibly the most important lesson that can be learned from this book is *how* transformational planning techniques can restrict the set of plans in ways that enable robot controllers to use simpler and more e cient planning algorithms.

Besides the computational model itself, the tools used to implement it are the main contributions of this research.

1. **Declarative Statements and Transparent Reactive Plans**
   The rst tool is an extension to the reactive plan language RPL that allows for an explicit representation of robot's intentions and beliefs. Reactive plans are not very transparent; thus, the ways in which the planner can reason about them are limited. The structure of the plans and the purpose of the subplans have to be represented in a transparent way to simplify reasoning about RPL plans. To do this, we added declarative statements for goals, perceptions, and beliefs to RPL.
   Declarative statements are important for plan revision. Goals can be used to analyze reasons for unachieved tasks, among them imperfect control, task interference, and external events. Perception statements are important because sensing is imperfect and faulty sensor data may cause plan failures. Belief statements can be used to identify incorrect control decisions and wrong instantiations of tasks based on incorrect beliefs that cause plan failures.
2. **Constructs for Locally Planning Ongoing Activities**
   The second tool is an extension to the reactive plan language RPL that allows for local planning of ongoing problem-solving behavior. The constructs provided by this extension allow routine activities to be planned in a flexible and focused manner: routine activities can reflect on how to perform its activities while executing them, focus on critical aspects of important subactivities, and ignore irrelevant aspects of their context. In addition, they can postpone planning when they lack necessary information or reconsider their course of action when noticing opportunities, risks, or behavior flaws. Finally, they can integrate revisions for activities smoothly while these activities are being performed.
3. **XFRML**
   The third tool, XFRML, is a concise notation for writing revision methods that allows planning processes to anticipate and forestall a wide range of behavior flaws that happen during the execution of routine activities. To diagnose the causes of projected behavior flaws, planning processes have to infer whether particular parts of the robot controller were executed and why. The use of XFRML makes such inferences possible because XFRML not only represents the physical e ects of executing a robot controller,

but also the process of plan interpretation, as well as temporal, causal, and teleological relationships among plan interpretation, the world, and the physical behavior of the robot.

By using XFRML and the declarative RPL commands I have implemented a failure taxonomy for the detection and analysis of plan failures that can occur in the performance of an achievement task. This failure taxonomy covers plan failures that occur when routine activities are carried out, perhaps with other jobs, by robots with imperfect sensors, imperfect control, an inaccurate and highly incomplete world model.

1.7 Outline of the Book

The remainder of the book is organized as follows. Chapter 2 lays the groundwork for the rest of the book. The chapter describes a simulated robot and the world it acts in. After the characterization of the physical constraints we introduce our model of reactive robot control, and describe the structure of reactive robot controllers. Chapter 3 describes the computational structure of the planning processes that anticipate and forestall behavior flaws. Chapter 4 discusses how structured reactive plans can be represented such that comparatively simple and fast algorithm for failure diagnosis and plan revision can anticipate and forestall behavior flaws. Chapter 5 is about XFRML, an abstract and concise notation for implementing plan revision methods. Chapter 6 details the implementation of the failure detection, and diagnosis from execution scenarios and the process of revising structured reactive plans. The chapter also describes and discusses the failure taxonomy for our simulated environment. We also discuss several failure models and plan revision methods for them in more detail. Chapter 7 explains the extensions of RPL for locally planning ongoing activity. I describe how these extensions are implemented and how di erent models of interaction between planning and execution can be realized in extended RPL. Chapter 8 evaluates structured reactive controllers as means for controlling autonomous robots, details their empirical analysis, and contrasts them to related research. The last chapter is the conclusion. I explain what makes structured reactive controllers work, list the main contributions of this book, and point out directions in which this line of research can be extended.

2. Reactivity

Robots have particular roles in their environments [130]. The current and future roles of robots include nuclear waste removal, o ce and hospital courier service, carpet cleaning in buildings, household tasks, spraypainting in automatic factories, and planetary exploration. Ful lling such di erent roles requires specialization in robot hardware: a robot acting in rough terrain needs legs whereas wheels su ce for an o ce robot. Like the hardware, the control software should also to some degree be tailored to speci c roles and environments. A planetary-exploration robot should choose its actions di erently than an o ce courier or a manufacturing robot. The planetary explorer has a xed mission in which the robot's safety is one of the highest priorities but it does not need to execute its activities very e ciently. An o ce robot, on the other hand, must accomplish diverse jobs in a more fast-paced environment in which it is not the only agent. It is unrealistic to expect one kind of robot controller to work well in such di erent jobs and environments. Moreover, many di cult robot control and perception problems can be solved more easily in speci c habitats [98, 97], especially when the habitats are designed to facilitate jobs [3]. To summarize, the desirable features of a robot controller are to a large degree determined by the capabilities of the robot and the features of its jobs and environment.

This chapter describes the role, environment, and physical capabilities of our simulated robot, which impose important constraints on the desired behavior of the robot and the capabilities of the robot controllers. It discusses concepts that support the implementation of routine activities concentrating on the reactive aspects of the robot's behavior. Reacting appropriately requires the robot to deal with unknown and changing aspects of the environment, respond to asynchronous events, and carry out di erent threads of control concurrently. Furthermore, the chapter demonstrates how these concepts can be realized in RPL (Reactive Plan Language) [124], the programming language used to implement the robot controllers described in this book. After discussing the individual control concepts we show how the di erent concepts can be e ectively combined within a *structured reactive controller*, which serves as a programming framework to realize the robot's routine activities [22].

2.1 The DeliveryWorld

Although technology is progressing fast [54, 105, 153], we still cannot equip robots with the repertoire of sensing and action routines that they need to accomplish daily activities reliably and time-e ectively. Perceiving the necessary and relevant information in real time is another serious obstacle. Therefore, we develop structured reactive controllers for a simulated robot acting in a simulated world. The simulated world and robot allow us to investigate issues in robot control by measuring how well and time-e ectively the robot accomplishes its jobs when run by di erent kinds of robot controllers.

Developing robot controllers and testing them only in simulated worlds is a risky enterprise. We will not know whether the controllers we build are successful until they run on real robots. Therefore, it is important to minimize the risk that the results obtained are irrelevant. To do this world simulators have to be designed carefully, along those dimensions we are interested in, so that they challenge robot controllers in the same way as real environments and robots would [94, 92] and that they are justi ed with respect to the current state of robotic research [65]. On the upside, world simulators allow researchers to run, debug, and test robot controllers for thousands of hours, a process necessary for the development and evaluation of software of this complexity. Another advantage of world simulators is that the parameters of the experiments are under the control of the experimenter: we can control exactly the degree to which slipping wheels and dirty lenses a ect our experimental results. This section describes the DeliveryWorld, the simulated world and robot that are used as the testbed for my research.

2.1.1 The World

Let us rst describe the important features of our world simulator in the DeliveryWorld and explain how it challenges robot planning techniques in realistic ways. Currently, xfrm does errand planning in a simulated world of discrete locations developed by McDermott and briefly described in [127].

The world is a grid of locations; each location has an in nite number of positions and at each position there is at most one object. A signpost (the object at position 4 in Figure 2.1) gives the coordinates of each location. Thus in order to nd out the coordinates of its location the robot can look for the numbers on the signpost. Locations can have other features like being wet, having distinctive surface characteristics, and so on.

Boxes, blocks, and balls of di erent colors, textures, and sizes, are spread out and can move, appear, disappear, or change their appearance in the face of exogenous events. Objects have prede ned features some of which can be perceived, among them visual characteristics but no names. Usually they cannot be identi ed solely by their perceivable features.

The complexity of the world simulator poses a serious challenge. The world contains a large number of objects (often between fty and one hundred

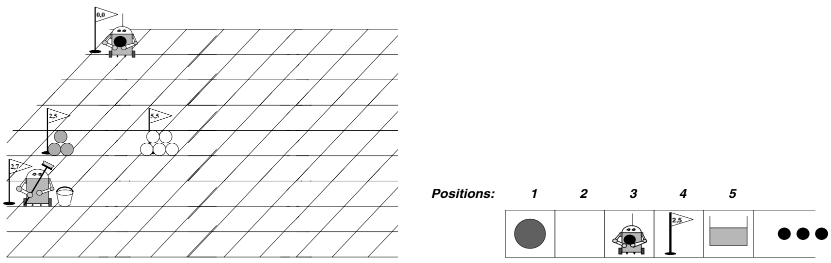


Fig. 2.1. Simulated grid world (left) and the objects and their positions at the robot's location (right).

objects plus four hundred signposts) with many features, most of which are irrelevant for the robot's jobs at a particular instant. This complexity makes it very hard to project and represent all changes in the environment as a plan gets executed. The world is also large compared to the robot. At any time the robot can sense and a ect only a small part of the world, namely the objects at its own location. These challenges are also characteristic of real environments like o ces or factories.

Another dimension of complexity is added by additional autonomous robots that inhabit the world and perform di erent kinds of jobs like cleaning locations or delivering and modifying objects. These robots are objects in the world that have a name and a job that they are carrying out. They can be sense each others within a certain vicinity and communicate the jobs they have. While robots do not work against each other their jobs might still interfere and the robot controllers might have to adjust to these interferences.

The simulated world also generates random events. An event has an average time of occurrence, a condition under which it happens, and a speci cation of the event e ects. So let us postulate an event that white balls disappear from locations with black pyramids every ve minutes on average. Whenever the world simulator is about to advance time by an amount t, it randomly decides whether a white ball disappears based on the probability of such an event occurring in an time interval of length t. If the simulator has decided that such an event *ev* has occurred, then it randomly determines a time t□ at which the event occurs. It must be between the current time and the current time plus the time step t. The world simulator then executes the body of the event description and continues.

Exogenous events and other robots cause changes in the world that are not under the control of the delivery robot. Since exogenous events are randomly generated the robot can have at best probabilistic models of them. From the point of view of the robot the world is nondeterministic. Besides making prediction more di cult, exogenous events and other robots require the delivery robot to react to unplanned events as they occur. Events occur asynchronously to the actions of the robot. For example, an event randomly blocks and unblocks locations and thereby requires the robot to circumnavigate the blocked locations.

In our simulator time passes at a realistic speed: that is, realistic estimates are used for the time it takes the robot to perform sensing and e ector actions. Time has three important roles in our experiments. First, the behavior of the robot is judged with respect to the time taken to accomplish the jobs. Second, the longer planning and execution takes, the more likely it is that relevant aspects of the world change due to exogenous events or the actions of other robots. Third, time determines the impact of planning on the behavior of the robot. If planning is too slow, revisions are proposed to parts of the plan that are in progress already. In these kinds of environments planners that are faster and do not produce optimal plans may produce a better performance than slower ones that produce more competent plans (see, for example, [138, 94]). Consider the following robot controller:

```
DO CONCURRENTLY
    1) go to  4,5 ; then execute plan FOO
    2) determine a better plan for FOO
END DO.
```

The robot controller consists of two concurrent threads of control. The rst one asks the robot to go to location ⟨*4,5*⟩ and then execute the plan FOO. The second branch tries to improve the plan for FOO. If the planning process comes up with a solution after FOO is completed, the planning has no e ect on execution. If planning nds a better plan for FOO when FOO is already partly executed, there may be costs incurred by adopting the new plan. Some of the accomplishments achieved under the old version possibly cannot be exploited by the new version. It could even be more advantageous to complete the old inferior plan than switching to the new more competent one (see, for example, [80]).

So far we have only considered the properties of the DeliveryWorld that make the robot's job di cult. On the other hand, the DeliveryWorld has a lot of structure that can be exploited to make the robot's jobs simpler. An important aspect is that for a delivery robot a comparatively few types of objects and world states are relevant. The objects include all types of objects that are to be delivered, boxes, other robots, and locations. States include those that indicate exogenous events, unreliability of vision routines, and exceptional states for manipulations. These characteristics correspond to results obtained in the computational modeling of the behavior of sh [157]. In order to generate realistic behaviors of sh, sh only have to distinguish very few categories of objects: sh of its own species, predators, food, and obstacles although underwater worlds are very rich.

Another factor that eases the accomplishment of jobs drastically is that exogenous events occur rarely. Also, the probability that another robot is performing jobs at the same locations as the delivery robot is quite low. Thus, routine plans that assume the absence of exogenous events and other robots have a good chance of being successful. However, they must watch out for clues that signal exceptional states. One kind of exceptional state are other robots acting in the neighborhood of the robot. Again the same kind of behavior can be found in animals: even when occupied with a particular activity, animals generally notice and orient towards new features in their surrounding environment [84].

The time that is needed to move around and manipulate objects often gives the robot enough computation time to adjust to exceptional states by means of planning.

In general, the study of the interactions between agents and their environments has become an important research e ort that tries to exploit structure in the environment in order to simplify the computational tasks to be performed by the agents. A good overview on this research e ort can be found in [4].

2.1.2 Commands and Jobs

The properties of the commands and jobs given to the robot also determine desiderata for the robot controllers to be used. Our delivery robot is taskable: it is supposed to carry out multiple jobs, which can be changed by adding, deleting, or modifying jobs at any time. The jobs are instances or variations of more general jobs that the robot has done often before. The robot has routines for carrying the general jobs that can handle many but not all variations of possible job instances. The following is a typical set of jobs:

1) get all black blocks from $\langle 1,13 \rangle$ to $\langle 1,4 \rangle$
2) get all boxes from $\langle 1,13 \rangle$ to $\langle 4,9 \rangle$
3) paint all pyramids at $\langle 1,4 \rangle$ black
4) get the white shiny block from $\langle 4,9 \rangle$ to $\langle 4,3 \rangle$

The robot's individual jobs are diverse. They consist of achieving a state of a airs, like getting objects to particular locations, maintaining a state, or getting information. More speci cally, the robot can be asked to transport, collect, change objects, or keep them at speci ed locations. Jobs can be quanti ed, as in *get all boxes from* $\langle 1,13 \rangle$ *to* $\langle 4,9 \rangle$, or refer to single objects, as in *get the white shiny block from* $\langle 4,9 \rangle$ *to* $\langle 4,3 \rangle$.

Sometimes a job cannot be accomplished without taking into account the remaining jobs of the robot. Jobs might be incompatible and require the robot controller to infer possible interferences between them in order to accomplish all of them reliably. If the robot received the jobs *(1) get all black objects from* $\langle 1,13 \rangle$ *to* $\langle 1,4 \rangle$ and *(2) get a block from* $\langle 1,13 \rangle$ *to* $\langle 1,4 \rangle$ it might pick one of the objects delivered in the rst job which is not the intention of the supervisor, the person giving the commands. Interferences are rarely so obvious and more often are buried deep within the routines.

The objective of the robot controller is to carry out the given jobs reliably and cost-e ectively. The jobs specify conditions on the e ects that the robot should have on its environment. If the supervisor commands the robot to deliver an empty box to a speci ed location, he cares about whether there will be an empty box at the location when the robot is done and how fast the box will get there. An experimenter can assign the robot a score for a given period based on the number of jobs it has performed, the degree to which it has accomplished them, and the time it has spent.

2.1.3 The Robot

Let us turn to the physics-based model of the simulated robot and its basic control routines. The robot is equipped with two hands, two cameras, and a radio link. The cameras allow the robot to look for objects at its current location, track them, and examine them. Performing each of these activities takes time and blocks resources: while looking for objects the robot can neither move to another location nor use the camera for another task. Vision is limited to the area within sensor range. Since even this area is complex the robot does not try to perceive whole scenes but actively looks for the information needed for executing its routines [8, 12, 13]. Thus, when activating a vision routine, the robot controller parameterizes the routine with the information it wants to extract from an image. For instance, the routine LOOK-FOR can be called with red balls as parameters. This routine returns a possibly empty set of designators, data structures that partially describe the red balls and their positions in the image. Since the robot's cameras are imperfect, designators will be inaccurate and the robot may overlook or imagine objects. If the controller has a designator that describes an object in the current image it can use this designator to track the object.

Our model of robot vision makes two important assumptions: that the vision routines can recognize objects satisfying a given symbolic description; and that the robot only sees what it is looking for. The rst assumption is necessary even without taking planning into consideration. The jobs given to the robot are formulated in a symbolic language. When given the job paint the red ball, the vision routines have to conclude that some part of the camera image is produced by a red ball or signal a failure if they are unable to do this. Also, we assume that the robot has visual routines to distinguish between objects that are relevant for its jobs [160]. Firby et al. [74] describe vision routines for trash collection, which detect and discriminate floor objects. The routines work under the assumption that small objects are trash and large objects are waste baskets. While this approach is necessary to interpret visual information in real-time, it yields the problem that the robot may not be able avoid a rapidly approaching robot because it is looking for something else. This problem can be avoided by running fast concurrent routines that detect approaching objects.

The DeliveryWorld simulator allows for the de nition of logical cameras or models of perception processes running on a particular camera. For instance, a logical camera for the recognition of objects with given features can be speci ed by de ning the following functions:

Objects within sensor range: returns all objects in the sensor range of the camera. Currently the sensor range is the robot's current location.
Objects seen: predicate that returns true if an object is seen. The predicate can be implemented probabilistically and depending on the distance to the robot and the properties of the surroundings.
Imagined objects: list of objects that are imagined by the robot sensors.
Perceived properties: given an object and a list of properties, the function returns the list of perceived property values. The perception can thus be noisi ed; often the color of wet red balls is perceived as violet.

To grasp or manipulate an object, the robot must track the object, that is it must keep the camera pointed at the object to get a visual feedback for controlling the robot's hands. In addition to recognizing objects with given features and tracking, the cameras can examine objects, i.e., extract additional visual features of a tracked object. It goes without saying that only one process can control and use a camera at a time and that threads of control might have to wait for access to the camera.

Robot localization and navigation in the DeliveryWorld are greatly simpli ed by the fact that the world is a grid of locations and that each location has a signpost showing the location's coordinates. Despite the simplicity we can generate place recognition problems by de ning as noisy the logical sensor LOOK-FOR-SIGNPOST. A more realistic model of place recognition and navigation is described in [65]. Besides cameras, hand force sensors that measure the force applied by the robot's grippers are the only other sensors the robot uses. If the hand is holding an object this force measure is high otherwise it is low.

To change its environment, the robot can reach out a hand to the positions of tracked objects, grasp them, and pick them up. The robot can also move around and manipulate objects by painting or cleaning them. The robot can only manipulate objects at its own location and has to track the objects and its hands while manipulating the objects. Acting is also imperfect. While grasping or carrying objects, objects might slip out of the robot's hands or might be stuck when the robot opens a hand.

2.1.4 Justi cation of the DeliveryWorld

The following list enumerates the features incorporated in the world simulator in order to secure that the simulator is not too simple to be an e ective tool for investigating autonomous robot control and that the results obtained in the investigation are relevant for real robots:

1. **Time proceeds at a realistic pace.** Time proceeding at a realistic pace presents several important challenges for robot planning. First, it limits the time in which planning can contribute to the behavior of the robot. It also requires the planner to balance the improvements that can be achieved with the time necessary to achieve them. Second, time costs

are part of the measure for the problem-solving behavior. Finally, parts of the world model used for planning may increasingly diverge from the real state of the environment due to changes in the environment that the robot has not perceived yet.

2. **Many realistic misbehaviors of real robots occur in the simulated world.** Another realistic challenge posed by the world simulator is the variety of causes for misbehavior that are generated by the world simulator. The sensors are imperfect and therefore can overlook or imagine objects, or misperceive features of objects. Many kinds of failures in place recognition and navigation and their consequences can be generated by noisifying the robot behaviors that look for and read signposts. The active vision behaviors produce only partial descriptions of the objects they identify and therefore these descriptions might be ambiguous with respect to the objects they refer to. Furthermore, the modules for controlling the hands are imperfect and may therefore not perform in the desired way. Other failures might occur because of changes in the environment caused by events not under the control of our robot. Various failures that are caused by di erent kinds of interferences among subroutines also occur in the simulated robot behaviors.
3. **Control programs have to be su ciently flexible and robust to perform jobs reliably.** Control routines must be concurrent reactive processes that adapt very flexibly to the situations they confront. Our world simulator generates asynchronous events and therefore our robot has to monitor aspects of its environment continually and react immediately to unexpected events. It must also be able to recover locally from detected plan failures.
4. **Many assumptions can be justi ed by current robotic research.** The primitive control routines that drive our robot are implemented in the same way as the skills used by a real robot for trash collection [74]. This robot uses skills like nding a small (large) object on the floor, turning the pan tilt head to track the target location, and moving to the target while avoiding obstacles.

Although we think that the above arguments justify that we can expect the results obtained in the world simulator to be relevant for real robots, the ultimate test is to run the kind of controllers we propose on real robots. An extensive discussion of the role of testbeds and empirical evaluation of agent architectures can be found in [94].

2.2 The Implementation of Routine Activities

This section analyzes implementation strategies for delivery routines and describe how these strategies can be implemented in RPL [124]. RPL programs look very much like LISP programs that make use of control abstractions typical of structured concurrent programming languages. Such abstractions include those for sequencing, concurrent execution, conditionals, loops, assignments of values to program variables, and subroutine calls. Several high-level concepts (such as interrupts and monitors) are provided; they can be used to synchronize parallel actions to make plans reactive and robust.

2.2.1 Plan Steps vs. Concurrent Control Processes

To explain why we want the plans controlling the robot in the DeliveryWorld to be collections of concurrent control processes rather than sets of discrete action steps, let us start by looking at the organization of layered hybrid robot controllers (see section 1.6) that make the assumption that at some level of abstraction the behavior of the robot is su ciently described by a partially ordered set of discrete steps. This is the level of abstraction at which planning takes place. Reactive and sensor-driven behavior is speci ed within the steps. Thus, at the planning layer, the controller for a delivery deliver object A, which is at location ⟨*0,3*⟩, to location ⟨*4,6*⟩ could be speci ed as a sequence of four steps: (1) (GO *0,3*), (2) (PICK-UP A), (3) (GO *4,6*), and (4) (PUT-DOWN A).

In the DeliveryWorld going over large distances takes considerable time. It is therefore more e cient to carry out steps opportunistically, when the robot happens to be at the location where a given step should be taken. However, steps like go to a particular location are too coarsely grained to exploit this kind of opportunism. Therefore, it is necessary to break up these steps into sets of (MOVE *DIR*) steps that bring the robot to the neighboring grid cell in direction *DIR*. In this case, the delivery plan becomes a set of MOVE steps, the PICK-UP step, another set of MOVE steps, and nally the PUT-DOWN step.

Where locations are not blocked, the order in which MOVE steps are carried out does not a ect the robot's nal position. If locations are blocked, the robot planner computes and carries out optimal navigation plans, which are disjunctions of *partial orders on* MOVE *steps* that bring the robot to its destination without sending it past blocked locations. If, however, the robot does not know in advance which of the locations are blocked, it will compute optimal paths to the destination and execute them until it runs into a blocked location. After running into a blocked location it will stop, plan a new optimal path to circumnavigate the blockage, and start the execution of the new navigation plan. This stop and replan strategy is often ine cient: the robot has to wait for the termination of the planning e ort before it can take its next step.

Hierarchically structured plans seem to resolve these problems: at a high level of abstraction they ask the robot to go to ⟨*0,3*⟩ abstracting away from the details of how to get there, while at a lower level they specify the series of move operations to get there. However, even at the higher level the planning process cannot abstract away from the route the robot takes to get to its destination. In the DeliveryWorld the robot might have or want to carry out actions that are triggered by observations that the robot makes on its way. Such actions for handling opportunities and risks and synchronizing with other robots often have important e ects on the intended course of action, even at the highest level of abstraction.

My conclusion is that abstracting away from the routes the robot takes oversimpli es the prediction problem if important actions are triggered by the robot's observations on its way. It is also counterintuitive to make strong commitments to inflexible navigation plans when the robot's information about the world is incomplete or unreliable. Therefore, I propose that the planning system should reason about flexible and robust navigation procedures or robot control programs in general, even if these flexible behaviors do not specify the optimal behavior. Like navigation, other activities of the robot, as the search for a particular kind of object or the exploration of the world, cannot be adequately described as partially ordered sets of discrete action steps.

Thus, the alternative is to reject the idea of having a layer of control where the behavior of the robot can be completely described as a partially ordered set of steps. In this approach, behavior will always be the result of concurrent control processes with partial orders as a special method for synchronizing the control processes. Going to a particular location could then be the result of two concurrent control processes: the rst causes the robot to go towards the destination until it has arrived there and the second causes it to circumnavigate blocked locations whenever it runs into them [114]. Opportunities like passing the location where an object has to be painted can be exploited if the go to destination process can be interrupted and resumed afterwards. Another advantage of this approach is that behaviors like exploration, search, and delivery can be implemented as variants of the go towards behavior by adding concurrent control processes to the go process. For example, if the purpose of going to location $\langle 4,6 \rangle$ is to carry an object, it will make sense to activate another process that monitors the force sensor of the robot's hand such that the robot can notice when it loses the object while going to the destination. If the object has been dropped, the robot should immediately stop, look for the object, pick it up, and then continue to $\langle 4,6 \rangle$. Other processes that are not part of the delivery such as watching out for opportunities and risks should also be active.

The speci cation of robot behavior using concurrent control processes with di erent priorities is a powerful technique to deal with partly unknown and changing situations [35]. One can specify processes that monitor both the progress of execution and important aspects of the environments, recover from local problems and react to asynchronous events.[1]

[1] Note, most robotics researcher have moved away from the pure reactive-control paradigm because it lacks concepts for structuring control programs (see, for example, [152]).

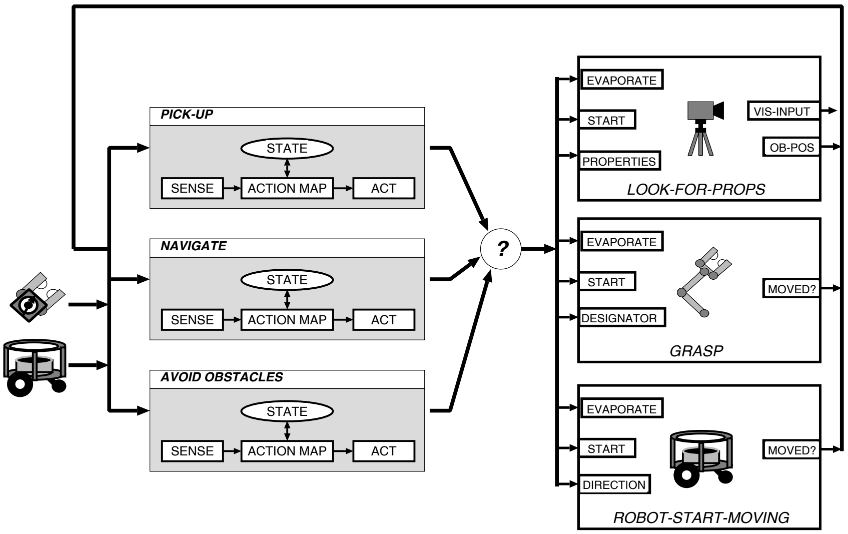


Fig. 2.2. Implementation scheme for routine activities in structured reactive controllers.

The implementation scheme for structured reactive controllers distinguishes between two kinds of control modules: continuous control processes such as grasping and moving and concurrent control routines. The concurrent control routines specify how the robot is to respond to feedback from the continuous control processes and sensor data in order to accomplish its jobs (see gure 2.2). The purpose of executing control routines is to activate and deactivate control processes, react to asynchronous events, and monitor the execution of the processes. The di cult part in the implementation of such controllers is the speci cation of the interactions among the concurrent control routines so that they cause the robot to exhibit the intended behavior.

2.2.2 Interfacing Continuous Control Processes

To facilitate the interaction between the concurrent control routines and the continuous control processes RPL supports a uniform protocol for the interaction between RPL programs and the control processes. The control abstraction that implements this protocol is called a *behavior module.*

Figure 2.3 shows the behavior module LOOK-FOR-PROPS. LOOK-FOR-PROPS can be activated with a perceptual description of the kinds of objects the robot is looking for (red balls, for instance), and the camera it is supposed to look with. The control process that is performed by the behavior module takes a camera image, searches for red ball-like objects in the image, and stores the position of each red ball in the register OB-POS, a location in the memory.

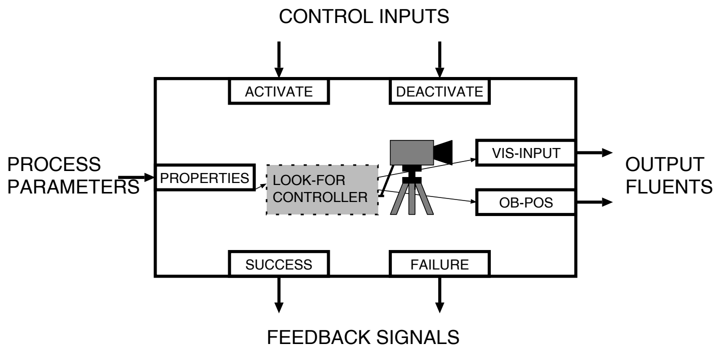


Fig. 2.3. The behavior module LOOK-FOR-PROPS.

RPL programs can a ect the execution of control processes in two ways only: by activating and parameterizing a control process and by deactivating it. Of course, on deactivation control processes cannot simply stop. The robot must not stop the execution of a process cross the street while it is in the middle of the street. RPL provides a construct (EVAP-PROTECT *A* *B*) that secures the execution of the code piece *B* in the case that *A* evaporates in the midst of its execution.

Control processes recognize the completion of the desired behavior and its success [127, 72]. For instance, the successful completion of a grasp is detected if the force measured by the hand force sensor exceeds a certain threshold. A failed grasp can be detected if ngers have minimal distance to each other while the hand force is still zero. In this case, the control routine signals an execution failure along with a description of the failure. Detecting the termination of actions boils down to discretizing continuous processes [71].

The last issue in the implementation of behavior modules that must be discussed here is the way the control processes communicate with the RPL control routines. RPL distinguishes between two kinds of communication: one within a thread of control and the other between di erent threads of control. Within a single thread of control, behavior modules signal an execution failure or the successful completion of the behavior and possibly return values on their termination. For example, the lter of LOOK-FOR-PROPS that searches the image for objects satisfying the given perceptual description, sends a success signal and the set of designators describing the matching objects back to the activating thread of control. Failure signals are sent if another behavior takes over the camera in the midst of the control process. However, communication within the thread of control is often not enough. Other threads of control might wait for the robot to arrive at a certain location or for the completion of a vision process. Control processes update global registers to notify other control threads about their execution status. For example, the behavior module LOOK-FOR-PROPS pulses the fluent VIS-INPUT in order to signal that the visual search is completed.

2.2.3 Coordinating Control Processes and Reacting to Asynchronous Events

Using the concept of behavior modules, we are now able to talk about patterns of control that allow a controller to react to asynchronous events, coordinate concurrent control processes, and use the feedback from control processes to make the behavior of the robot robust and e cient. RPL supports the coordination of concurrent control processes by providing means to declare threads of control to be RPL processes and means for connecting control processes to sensors and e ectors. RPL also provides concepts for interprocess communication via shared variables.

To realize the go to destination behavior in our example, a programmer might implement two control processes go towards destination and circumnavigate blocked location that work as follows. The sonars for detecting obstacles should be read frequently and, as soon as an obstacle is detected, the CIRCUMNAVIGATE process should interrupt the GO-TOWARDS process and get the robot around the obstacle. Then the GO-TOWARDS process should continue. To achieve this kind of behavior a programmer needs to

1. connect RPL control threads to sensors,
2. trigger processes with asynchronous events, and
3. resolve conflicts when active control processes send conflicting commands to the robot's e ectors. In our case, the processes go towards destination and circumnavigate blocked locations might cause a conflict by asking the robot to go into di erent directions.[2]

Connecting Control Threads to Sensors. In order to connect RPL control threads to sensors and allow them to react to asynchronous events one can make use of *□uents,* speci c types of program variables that can signal changes in their values. Fluents are best understood in conjunction with the RPL statements that respond to changes of fluent values. For instance, the RPL statement (WHENEVER *F B*) is an endless loop that executes *B* whenever the fluent *F* gets the value true. The statement is implemented such that the control thread interpreting it goes into the state wait-blocked as long as *F* is false; waits for the fluent to signal that it has become true; and then resumes the interpretation. Besides WHENEVER, WAIT-FOR and FILTER are other control abstractions that make use of fluents. The RPL expression (WAIT-FOR *F*) blocks a thread of control until *F* becomes true. (FILTER *F B*) executes *B* until the value of *F* becomes false. Fluents can be set and changed by sensors,

[2] Of course, conflict resolution and command arbitration are very complex. For instance, we might want the robot go around an obstacle into the direction of the destination (see, for example, navigation in Sonja [46] or navigation using potential eld methods [111]). The implementation of such navigation methods might require more complex interactions between the behaviors such as passing the destination as an argument to the circumnavigation process.

behavior modules, and the structured reactive plan. Examples of fluents that are set by the robot's sensors are HAND-FORCE-SENSOR, VISUALLY-TRACKED, and LOCATION-AHEAD-IS-BLOCKED. These fluents are used to write pieces of programs that react to changes in the environment and the state of the robot.

Behaviors, like avoiding obstacles or following another robot, are best described as reactive behaviors that try to reduce the di erence between a sensed state of the world and a desired one. Rather than constructing and maintaining world models, the reactive robot controllers sense the robot's surroundings at a higher frequency and map their sensor data directly into activation/deactivation signals for the control processes. A convenient implementation method for reactive behaviors are collections of condition-action rules. The condition-action rules select the appropriate actions for each sensor reading without deliberation. Consider, for example, the behavior required for following another robot, which has to maintain the distance and relative orientation to the other robot continuously [49]. The desired state is to be behind the other robot at a certain distance. The behavior has to read the sensor data frequently and the robot drive has to adapt its speed and heading frequently. The mapping from sensor data to actuator activations and deactivations can be quite simple. The sensed states for distance could be too far, ideal, and too close, for the relative lateral position of the other robot left, ahead, and right. The actions could be turn left, turn right, and go straight ; and increase speed, decrease speed, and keep current speed. The following piece of RPL code is an example of how this style of control can be implemented in RPL:

```
(PAR (WHENEVER NEW-DIRECTION-READING
            (TRY-IN-ORDER
                ((= RELATIVE-LATERAL-POS LEFT)
                 (ACTIVATE-TURN-LEFT))
                ((= RELATIVE-LATERAL-POS RIGHT)
                 (ACTIVATE-TURN-RIGHT))))
      (WHENEVER NEW-DISTANCE-READING
            (TRY-IN-ORDER
                ((= DISTANCE TOO-FAR) (INCREASE-SPEED))
                ((= DISTANCE TOO-CLOSE) (REDUCE-SPEED)))))
```

Behavior Composition. The question remains as to how the processes GO-TOWARDS and CIRCUMNAVIGATE should be synchronized. The desired behavior is that the robot goes towards its destination. Then, if it is heading towards a blocked location, the circumnavigate process interrupts the GO-TOWARDS process, gets the robot around the blocked locations and returns control until the robot is facing the next blocked location. This is accomplished by the following piece of RPL code:

```
(WITH-POLICY (WHENEVER LOCATION-AHEAD-IS-BLOCKED?
                  (CIRCUMNAVIGATE-BLOCKED-LOCATIONS DEST))
    (FILTER (NOT (= ROBOT-LOCATION DEST))
       (GO-TOWARDS DEST)))
```

In this example we make use of another RPL construct: (WITH-POLICY P B), which means, execute B such that the execution satis es the policy P. In our case, go towards the location *DEST* until you are there; is subject to the policy that whenever the robot is heading towards a blocked location go *rst* around the blocked location before you continue to go towards the destination. Thus the WITH-POLICY statement can be used to implement the vital behaviors of the robot by turning them into the policies for the statement.

Other control structures that combine plans into more complex activity descriptions are the RPL statements PAR, SEQ, LOOP, and IF. The statement (PAR P_1 ... P_n) speci es that the plans $P_1, \ldots, P_n$ are to be executed in parallel. (SEQ P_1 ... P_n) asks the robot to execute the plans sequentially. (LOOP P UNTIL C) executes the plan P repeatedly until C is satis ed. (IF C P_1 P_2) performs P_1 if C is true, P_2 otherwise.

2.2.4 Synchronization of Concurrent Control Threads

We have just explained how RPL plans can be connected to sensors and how behaviors can be combined into more complex behaviors. We will now describe additional constructs for the synchronization of control threads that support the realization of routine activity.

Synchronizations, such as preventing the robot from starting toward its destination before it has an object in its hand, can be speci ed using statements like (WAIT-FOR *(> HAND-FORCE 0.5)*). To support the synchronization between concurrent control threads, the interpretation of each piece of RPL program P is also associated with two fluents: (BEGIN-TASK P), which is pulsed when the interpretation of P starts, and (END-TASK P), which is pulsed when the interpretation of P is completed. These fluents can be used to constrain the order in which two threads of control are to be carried out.

When running many control processes concurrently another problem is often encountered. Di erent processes may issue incompatible commands to the camera, the motor, or the hands of the robot and thereby interfere with each other. The problem of resolving conflicts when multiple control processes try to control the same e ector simultaneously in di erent ways is called *task arbitration*. Intuitively, our robot controllers must satisfy the constraint that at any time one control process at most can use a camera or determine where the robot should go. The di culty is, of course, to determine which control process should have control over which resource. We implement this constraint by associating a semaphore, called a *valve*, with each e ector. The valve represents the right to issue commands to the corresponding e ector. For example, if a control process wants to cause the robot to move, the process must own the *WHEELS*, the valve associated with the robot's motor. Below is a piece of RPL code that does what we want:

```
(PROCESS GO-TO-DEST
    (VALVE-REQUEST GO-TO-DEST WHEELS)
    (FILTER (NOT (= ROBOT-LOCATION DEST))
      (GO-TOWARDS DEST)))
    (VALVE-RELEASE WHEELS))
```

The RPL constructs that allow for these kinds of synchronizations are (PROCESS *P B*), (VALVE-REQUEST *P V*), and (VALVE-RELEASE *P V*). (PROCESS *P B*) de nes a program unit *B* with name *P* that can own valves. When executing (VALVE-REQUEST *P V*) the process *P* asks for the valve *V*. In the example above the robot waits until it gets the valve *WHEELS*, and after it receives the *WHEELS*, they are blocked for all other control routines. On the robot's arrival at its destination, the wheels will be released, and other processes waiting for them can compete for control over them. If policies or more urgent processes need to take over the wheels they pre-empt the valve.

2.2.5 Failure Recovery

Besides sensor-driven and synchronized concurrent behavior, the recovery from execution failures is another important aspect of robust robot behavior. Sometimes control processes notice that they have not accomplished their tasks or cannot do so. The process for grasping an object may detect, on completion, that the measured hand force is still low. In this case the continuous control process will send a failure signal using the RPL statement (FAIL :CLASS *C* *ARGS*): for example, (FAIL :CLASS *UNSUCCESSFUL-GRASP DESIG*).

RPL provides several control abstractions that can be used to specify how to react to signalled execution failures. For instance, the statement (N-TIMES *N B* UNTIL *C*) executes *B*, ignoring execution failures, until condition *C* holds but at most *N* times. Or, the statement (TRY-IN-ORDER M_1 ... M_n) executes the methods M_1, ..., M_n in their order until the rst one will signal a successful execution. Using these two constructs we can implement a more robust routine for grasping an initially unknown object. There are di erent methods. If the object turns out to be a ball smaller than the hand span, the robot can wrap its hand around the ball from above. If the object is a block, the robot can try to grasp it by putting its ngers on opposite sides of the block. If the object is a box larger than the hand span, the robot should try a pinch grasp. The plan asks the robot to grasp the object repeatedly until it has it in its hand; but to give up after failing for the third time. This plan fragment (which has a meaning very similar to that of RAPs proposed by Firby [70]) shows a reactive plan that selects di erent methods for carrying out a task based on sensory input and tries the methods repeatedly until it is successful or gives up:

```
(LET (SIZE CAT)
   (N-TIMES 3
      (!= SIZE (EXAMINE OB '(SIZE)))
      (!= CAT (EXAMINE OB '(CATEGORY)))
      (TRY-ONE
           ((AND (EQ CAT 'BALL) (< SIZE HAND-SPAN*))
            (WRAP-GRASP OB H 'FROM-ABOVE))
           ((AND (EQ CAT 'BLOCK) (< SIZE HAND-SPAN*))
            (GRIP-GRASP
               OB H (ONE-OF '(FROM-LEFT FROM-RIGHT
                              FROM-FRONT))))
           ((AND (EQ CAT 'BOX) (   SIZE HAND-SPAN*))
            (PINCH-GRASP OB H 'FROM-ABOVE)))
     UNTIL (FLUENT-VALUE HAND-FORCE*)))
```

2.2.6 Perception

Robot controllers cannot refer directly to objects in the real world. They have to use sensor programs that can look for objects matching a given description and compute *designators*, data structures that describe each matching object and its location and position. The robot controllers use these designators to nd and manipulate the objects they describe. To use designators for parameterizing control routines, the robot controller has to store them and pass them as arguments. This can be done using the LET- and the assignment statement (!= *V E*). The statement (LET $((V_1\ E_1)\ \ldots\ (V_n\ E_n))\ b$) creates local variables $V_1, \ldots, V_n$, initializes them with the values of the expressions $E_1, \ldots, E_n$ and executes B in the extended environment. The statement (!= $V\ E$) sets the value of V to the value of the expression E:

```
(LET ((DESIG-FOR-BALL FALSE))
   (!= DESIG-FOR-BALL
        (LOOK-FOR-A '((CATEGORY BALL) (COLOR RED)) CAMERA-1))
   (PICK-UP DESIG-FOR-BALL RIGHT-HAND))
```

In this example the robot controller uses a local variable with name DESIG-FOR-BALL to store the designator returned by the LOOK-FOR-A control routine. The PICK-UP routine is then called with DESIG-FOR-BALL as an argument. There are problems caused when the actions of the robot are speci ed using object descriptions rather than the objects themselves: the designators might not correctly describe the intended object, or they might be ambiguous, that is describe more than one object.

2.2.7 State, Memory, and World Models

To act e ciently the delivery robot in the DeliveryWorld must remember what it has seen. To resume its activities robustly after interruptions, it must remember where it put things. If the robot needs an empty box it should remember where it has seen one, because at any given time the robot can sense only its local neighborhood, and nding empty boxes without knowing where to look is time consuming. In all these cases, the memory needed comes in form of global variables. Many RPL control programs make use of global variables to store and maintain important information they have received by sensing.

The most important of these is the variable KNOWN-THINGS, which contains partial descriptions of objects the robot has seen so far. If the robot wants to use KNOWN-THINGS only to answer questions like where have I last seen an object that satis es *P*? it is su cient simply to add all designators obtained by sensing operations to KNOWN-THINGS.

However, if the robot wants to make more informed guesses about what might be at a certain location at the current time, it will not be enough to add all perceived descriptions to KNOWN-THINGS. If KNOWN-THINGS is intended to constitute a partial model of the state of the environment, it must be updated more thoroughly. As a world model, KNOWN-THINGS is necessarily incomplete because it only contains designators for objects already perceived. It may even be wrong since (a) the designators are produced by noisy sensors and (b) exogenous events may change the world without the robot's noticing.

A useful requirement for updating KNOWN-THINGS is that descriptions of the same objects should be fused. Suppose that there is a location with two black boxes and the robot goes there twice, once to look for boxes and once to look for black objects. If the update mechanism just keeps adding designators to KNOWN-THINGS the robot would think there are at least four objects at this location: two boxes and two black objects.

To avoid this confusion the robot controller needs to fuse results from di erent perceptions and revise older descriptions to include the results of the recent perceptions. The sensing routine LOOK-FOR can be run in two modes, one that updates KNOWN-THINGS and one that does not. The updating version works as follows. LOOK-FOR computes expectations for what to see, based on the current value of KNOWN-THINGS. It then compares what it expects with what it sees. If it sees more objects than expected, the designators for the remaining objects will be added to KNOWN-THINGS. If it sees fewer objects than expected, it will remove excess designators from KNOWN-THINGS. If the set of expected objects is non-empty, the update algorithm will try to fuse old and new descriptions. To make a decision about whether or not to fuse two designators, the robot might have to examine additional features of the objects in question. In our example the robot comes back to look for black objects and wants to update KNOWN-THINGS. The update algorithm requires the robot to examine the shape of *all* black objects, because otherwise it could not decide whether the black object at location *L* and the box at location *L* are actually the same object. An algorithm di erent from Firby's memory update algorithm was required because Firby's does not need to provide a single consistent view of a location [70].

2.2.8 The Structure of Routine Activities

There are several features of everyday activity for which we have not yet discussed any implementation methods. First, we have said that routine activities are skills associated with the problems they solve [156]. We have added the statement (DEF-ROUTINE-ACTIVITY *pattern body*) that can be used to specify pattern-directed routine activities. De ning a routine activity generates an entry in the plan library that can be indexed by *pattern*.

The following code example shows a typical RPL procedure for a routine activity. The routine rst computes the controller's estimation of the robot's position. If the position estimate is the destination of the object the robot checks whether the object is already in one of its hands. If so, it drops the object, otherwise it carries the object to the destination. If the object descriptor speci es that the object is at its destination, the plan succeeds if it can sense the object at the destination, otherwise it calls the subroutine CARRY-OB-TO-LOC. We call these kinds of reactive plans universal because they are designed to achieve their tasks no matter what the initial situation looks like and despite hindrances [147]. They do so by assessing the current situation based on sensor readings and internal data structures and selecting subplans based on this assessment. Universal plans are not described adequately by causal models with preconditions and e ects. They are designed to always achieve their goal and can be assumed to be fairly robust against interference from other processes. The most important requirement is that the object descriptor is useful to the extent that it speci es su ciently accurately where the object is and how it looks.

```
(DEF-ROUTINE-ACTIVITY (ACHIEVE (LOC OB  X,Y ))
   (LET* ((< XH YH > (COORDS-HERE)))
     (IF (AND (= XH X) (= YH Y))
         (IF (IS hand (DESIG-GET OB 'POS))
             (LET* ((H (FIND-OB-IN-HANDS OB)))
                (IF H
                    (UNHAND H)
                    (CARRY-OB-TO-LOC OB X Y)))
             (IF (AND (= (DESIG-GET OB 'X-COORD) X)
                      (= (DESIG-GET OB 'Y-COORD) Y))
                 (TRY-IN-ORDER (SEE-OB-HERE OB)
                               (CARRY-OB-TO-LOC OB X Y))
                 (CARRY-OB-TO-LOC OB X Y)))
             (CARRY-OB-TO-LOC OB X Y))))
```

The organization and structure of routine activities di ers a lot from the way classical robot control systems are organized. Routine activities are behavior-based control systems that are composed from sets of concurrent behaviors without a centralized control [118]. Also, program units in routine activities are supposed to accomplish a particular task as, for instance, avoiding obstacles or delivering an object and do all the necessary sensing, e ector control, and failure recovery themselves. This is di erent from the classical organization of control systems in which program units that contribute to a particular functionality like sensing or e ecting are composed together.

An important consequence of the task-speci c modularization of routine activities is that the modules have a purpose that can be perceived by an observer just by looking at the behavior of the robot. In particular, the observer can determine whether a particular module works in a given situation by looking at the physical behavior and how the world changes. If the robot bumps into an obstacle then the problem lies in the obstacle avoidance module, the sensors it accesses or the e ectors it controls provided that there are no programming errors such as an inappropriate task arbitration scheme or a valve that has not been returned after its usage. In a classically organized control system the cause for the misbehavior could not be narrowed down; it could still be anywhere in the sensing, the planning, or the action module. In concurrent reactive control systems the causes of misbehavior are easier to localize. Later in chapter 4 we will exploit this form of organization in concurrent reactive systems by representing the purposes of the modules explicitly and thereby facilitating the automatic understanding of the structure of robot controllers.

2.3 The Structured Reactive Controller

In section 2.2 we have described and discussed robot control concepts that are necessary for building e cient and reliable robot controllers for the delivery application in the DeliveryWorld. Let us turn to the question of how structured reactive controllers should be organized and the di erent concepts be used. Designing an appropriate architecture for structured reactive controllers serves two purposes. First, it facilitates writing control routines that can be used within the structured reactive controller; second, it makes it easier for planning processes to reason about and transform structured reactive controllers.

The *structured reactive controller* (see gure 2.4) controls the behavior of the robot based on incoming sensor data. Its main components are the *structured reactive plan*, the *planning* and *behavior modules*, global *uents* and *variables*, the *plan library*, and the RPL *runtime system*. In a nutshell the structured reactive controller works as follows. The RPL interpreter interprets the structured reactive plan, causing the behavior and planning modules to be activated and deactivated. Threads of control get blocked when they have to wait for certain conditions to become true. For example, if the robot is asked to go to a location L and pick up an object the controller activates the behavior of moving towards L. The interpretation of the subsequent steps are then blocked until the robot has arrived at L (i.e., till the move behavior signals its completion).

The architecture is an adapted version of the XFRM architecture [127]. The main di erence is that in XFRM there is a planning system that communicates

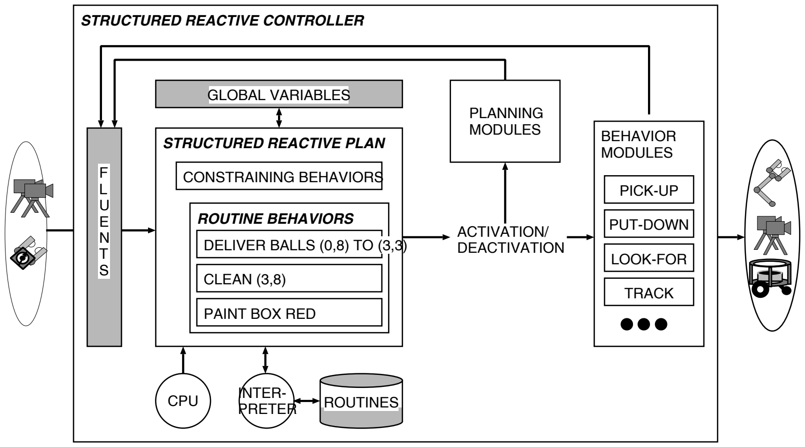


Fig. 2.4. Architecture of the structured reactive controller.

with the user and swaps plans. The architecture proposed here does not contain a planning system. Rather the structured reactive controller performs planning tasks by activating planning e orts in the same way as it activates continuous control processes.

2.3.1 Behavior and Planning Modules

The elementary program units in the structured reactive controller are the behavior and planning modules. Behavior modules constitute the interface between RPL programs and the continuous control processes that determine the physical behavior of the robot (see section 2.2.2 for a detailed discussion).

Planning modules (see gure 2.5) conform to the same protocol as behavior modules. They can be activated and deactivated by routine activities in the same way that behavior modules can be. Simplifying a bit, a planning module is activated with a routine activity as its argument and tries to nd a controller that is better suited for the current situation. A detailed discussion of planning modules and the interaction between planning and execution can be found in chapter 7.

Planning modules provide their results in the same way as behavior modules: they change fluents that can be read by other parts of the robot controller. Planning modules update three fluents: DONE?, which is true if and only if the planning module has searched every possibility to improve the plan it started with; BETTER-PLAN?, which becomes true as soon as the module has found a plan that is better than the one it started with; and BEST-PLAN-SOFAR, which contains the best plan the planning module has found so far. The planning module signals success when it has ended its search.

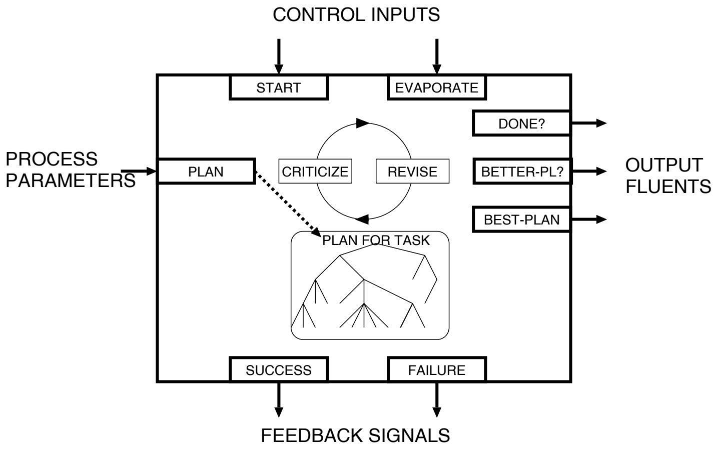


Fig. 2.5. Interface of planning modules.

2.3.2 The Body of the Structured Reactive Controller

The body of the structured reactive controller consists of *primary* and *constraining behaviors*. Primary behaviors are for the e cient and reliable accomplishment of top-level commands; constraining behaviors implement constraints on the execution of the primary activities. An example for such a constraint is that the robot has to avoid obstacles while performing its jobs.

Primary activities instantiate, activate, and deactivate behavior modules. Robustness is achieved by synchronizing the concurrent control processes, monitoring their execution, adequately reacting to unforeseen events, and recovering from local execution failures. To achieve the desired robustness and flexibility of everyday activity the structured reactive controller makes heavy use of control abstractions that can structure the complex everyday activities and make them reactive and adaptive (see section 2.2).

Constraining behaviors are processes that run all the time and have higher priority than the primary behaviors. They contain (1) vital behaviors, (2) monitoring behaviors, and (3) permanent behaviors. Most of the time these behaviors are wait-blocked, waiting for conditions to arise that require their intervention. For instance, the vital behavior *obstacle avoidance* is active only if the fluent set by the range sensors signals that an obstacle is in front of the robot. In this case the obstacle avoidance process interrupts all routine activities, moves the robot away from the obstacle, and goes back to sleep, and the routine activities are resumed. Note, routines have to be designed carefully to allow for such interruptability (see chapter 4.2).

2.3.3 Global Fluents, Variables, and the Plan Library

Global fluents and variables form the state of the structured robot controller. All structured reactive plans use the same set of global variables. The following table shows global variables and fluents used by our robot controller:

Global Fluents

HAND-MOVED*	Pulsed whenever the hand completes a movement.
HAND-FORCE*	Set to the force applied by the hand.
MOVE-INTENTION*	Pulsed whenever the robot is about to move somewhere else.
ROBOT-MOVED*	Pulsed whenever the robot completes a movement.
VISUAL-INPUT*	Pulsed whenever sensory information is available.

Global Variables

OB-POSITIONS*	Positions of objects recognized in an image.
X-REGISTER*	X coordinate of the odometer reading.
Y-REGISTER*	Y coordinate of the odometer reading.
KNOWN-THINGS*	Designators for previously perceived objects.

2.3.4 The RPL Runtime System

The remaining part of the structured robot controller is the RPL runtime system, which consists of a CPU and an *interpreter*. The CPU is the process management component of the structured reactive plan and activates, blocks, unblocks, and schedules the di erent threads of control in the execution of an RPL plan. Since planning processes install additional synchronizations and ordering constraints automatically, it often happens that these constraints are inconsistent and that two di erent threads of control block each other. Therefore, an important mechanism in the CPU of a structured reactive plan is deadlock breaking [127].

The RPL interpreter works as follows: it picks a thread, which speci es what to do and how to continue, from the enabled queue and interprets it. What has to be done is speci ed by the RPL code C contained in the thread. How the interpretation should be continued is speci ed by two sets of threads: one set contains the threads that should be executed when the interpretation of C succeeds and the other the threads to be executed if it fails. The threads returned by running a thread are then enqueued in the enabled queue. For example, the semantics of the RPL construct (SEQ $A_1 \ldots A_n$) is to execute the steps A_1 to A_n one after the other. The SEQ statement succeeds when the last step in the sequence A_n succeeds. If one step A_i fails, the SEQ-statement fails. Thus the interpreter handles a thread containing the statement (SEQ $A_1 \ldots A_n$), success follow-up threads S, and failure follow-up threads F as follows. It interprets A_1. If the interpretation of A_1 succeeds, it generates a new thread containing the statement (SEQ $A_2 \ldots A_n$) and the sets of threads S and F, and adds them to the enabled queue. If the interpretation of A_1 signals a failure it simply adds the threads F to the enabled queue.

2.4 Summary and Discussion

I have started this chapter by stating that the capabilities of robot control systems should depend on the characteristics of the environments the robots act in, the robots' jobs and their capabilities. This statement might seem noncontroversial. Still most AI research on the robot control and planning problem focuses on very general solutions to these problems, implicitly assuming that they work for wide ranges of environments, jobs, and robots. I believe that not tailoring the design of robot control systems to the speci- c needs of robots that are imposed by the environment, the jobs, and the capabilities of the robot will result in inferior robot control systems.

In this book we look at simulated robots that have limited and noisy sensing capabilities and imperfect e ector control. The robots are to accomplish multiple non-repetitive jobs in a changing and partly unknown environment. These characteristics are important because future autonomous service robots will have to solve robot control problems that share these characteristics. Thus, I test the structured reactive controllers with a simulated delivery robot in a simulated world, the DeliveryWorld, that has these properties. The important aspects of the DeliveryWorld that have to be accounted for by the design of structured reactive controllers are:

The world changes due to asynchronous exogenous events and other robots, and time passes at a realistic pace. Many aspects of the world that are relevant for its jobs are in the beginning unknown to the robot. The robot perceives a great deal of this information while accomplishing its jobs.

The delivery robot has to accomplish sets of complex jobs that require nding, delivering, and manipulating objects. The jobs can be changed any time. The objective of the robot is to accomplish its jobs as good as possible without wasting resources.

The delivery robot has limited and noisi ed sensing and imperfect action capabilities. It applies active vision routines that extract speci c information from camera images and take more time the more information is requested.

An important aspect of the DeliveryWorld that can be exploited by the robot control system is that the robot is given variations of the same jobs over and over again and confronted with typical situations many times. This aspect allows the plan writer to make the assumption that most activities performed by the delivery robot are routine, can be learned, and, once learned, do not require a lot of thought. Having robot control systems that can cope well with environments that have these features is important because future service robots such as autonomous o ce couriers will face the same kinds of challenges.

As I have stated in chapter 1, my approach to enabling the delivery robot to accomplish its jobs reliably and e ciently is through structured reactive controllers, collections of concurrent control routines that specify routine activities and can adapt themselves to non-standard situations by means of planning. Structured reactive controllers are adequate for robots that have to perform routine activities but also have to exploit information they learn during the plan execution.

Section 2.2 has explored concepts for the speci cation of reliable routine activities for the delivery robot. I have argued that routine activities are best speci ed by composing possibly concurrent behaviors rather than as are partially ordered sets of plan steps. In addition, I have shown that the robot control system of the delivery robot needs concepts to synchronize concurrent activities, monitor aspects of the environment while doing other things, locally recover from detected plan failures, avoid sending inconsistent commands to the sensors and e ectors, and locally store information about objects.

Another important observation is that the planning processes performed by structured reactive controllers cannot abstract away from routine activities being concurrent programs that specify how the robot is to react to sensory input without losing the capability to understand and forestall many possible behavior flaws caused by its routines. For example, to explain why the robot might have grasped the wrong object in a projected execution scenario, it might be necessary to look at an designator stored in a local program variable that should describe the intended object. Also, to avoid many behavior flaws the planner has to modify the control structures in the concurrent plans.

The tools that are provided to implemented structured reactive controllers can be grouped into three categories: tools for implementing robust routine activity, the transformational revision of routine activities, and for the speci cation of the interaction between planning and execution. In this chapter I have described the RPL system [124], consisting of the RPL language, its interpreter, and the runtime system as a very powerful and general tool for the implementation of routine activity. RPL provides many powerful control abstractions that have proven to be useful to make the behavior of autonomous robots with imperfect sensing and action capabilities acting in partly unknown and changing environments more robust. Tools for planning have been developed by McDermott [127, 128] and as part of this book (see chapters 4 and 5). Tools for specifying the interaction between planning and execution are described in chapter 7.

In the meantime RPL has been used to control two autonomous mobile robots: RHINO [42, 14] and Minerva [159]. An advantage of using RPL for autonomous robot control is that it enables robot controllers to integrate physical action [17, 22], perception [15, 10], planning [23, 20], map learning [27] and communication [26] in a uniform framework. This is possible because RPL allows for the speci cation of interactive behavior, concurrent control processes, failure recovery methods, and the temporal coordination of processes.

3. Planning

The selection of appropriate actions for the delivery robot in the DELIVERYWORLD cannot be solely based on the computational state of the structured reactive controller and the current sensor data. Often, the delivery robot has to make decisions as to whether it is advantageous to pick up a nearby empty box in order to use it later, in which order to accomplish jobs, or whether or not to spend e orts on making sure that a particular state remains true. Making the appropriate decisions requires the robot to look ahead of time and take expectations about future events into account. Such decisions require planning, more speci cally the revision of the course of action based on predictions.

This chapter addresses several aspects of planning. The rst part describes how structured reactive controllers are represented as syntactic objects that can be inspected and manipulated by the planner and describes RPL constructs that make plans more modular and revisions easier to install. The second part of the chapter is a description of the computational structure of transformational planning of structured reactive controllers. It describes the basic planning cycle, the computation processes, and the basic data structures. I will then show how the anticipation and forestalling of behavior flaws can be implemented as an instantiation of this transformational paradigm.

3.1 The Structured Reactive Plan

To reason about robot controllers, they have to be implemented as plans. Plans are maps, symbolic descriptions, of the future activity of the robot that can be analyzed, transformed, and executed.[1] Plans have two roles: rst, they are executable prescriptions that can be interpreted by the robot to accomplish its jobs; and second, they are syntactic objects that can be synthesized and revised by the robot to meet the robot's criterion of utility. These two roles of plans require any plan notation (1) to be capable of specifying the range of behaviors necessary for achieving the robot's jobs, and (2) to be su ciently transparent such that planning processes can predict

[1] Agre and Chapman [5] discuss alternative views of plans.

what might happen during the execution of a plan and transform the plan in order to improve it.

I have argued in chapter 1.6 that the representation of structured reactive controllers for planning should be the same notation as the one used for specifying how the robot is to react to sensory input: RPL. Therefore, a piece of RPL code is both, a plan as well as a control program: the plan is simply the *part* of the structured reactive controller that the planner explicitly reasons about and transforms. What distinguishes structured reactive plans from ordinary robot control programs are the kinds of operations that can be performed on them. Control programs can only be executed; plans can also be reasoned about and transformed. Thus what turns RPL into a language for representing structured reactive plans is the set of tools RPL comes with. Those tools enable planning processes to

project what might happen when a robot controller gets executed and return it as an execution scenarios [128, 127, 18];
infer what might be wrong with a robot controller given an execution scenario [127, 21, 24]; and
perform complex revisions on structured reactive controllers [127, 21, 24, 16, 25].

3.1.1 Plans as Syntactic Objects

For the purpose of planning RPL plans are internally represented as parse or *code trees*. Whenever the planner works with an RPL plan, it takes the plan text and converts it into a code tree that represents the structure of the plan. The nodes in the tree are code and subcode instances. The arcs are labeled with an expression that speci es how code and subcode is related. In RPL code trees all subcodes are nodes in the tree and have a name path that describes the path from the root of the code tree to the subcode itself.

For example, consider the tree for the code piece (PAR (IF C γ)) shown in gure 3.1. The subcodes of (PAR (IF C γ)) are , (IF C γ)), and and have the *code pre xes (BRANCH 1)*, *(BRANCH 2)*, and *(BRANCH 3)* respectively. The names of the branches depend on the type of the control structure. The subcodes of an IF-statement, for example, have the code pre- xes *(IF-ARM TRUE)* and *(IF-ARM FALSE)*. A code piece C is indexed by its *code path*, the sequence of code pre xes of the nodes from the root of the code tree to C. The *code path* for the code piece is *((BRANCH 2) (IF-ARM TRUE))*. In order to refer to a piece C of the plan the planner has to know the path from the root of the code tree to the subtree that represents C.

A transformation of a RPL program is implemented as a series of replacements of subtrees in the code tree. In order to do these replacements the planner uses the operation (CODE-REPLACE *CODE-TREE PATH SUBTREE*) [127]. Figure 3.2 shows the replacement of (IF C γ)) by (SEQ (IF C γ))) in the code tree representing (PAR (IF C γ)). There are two complications

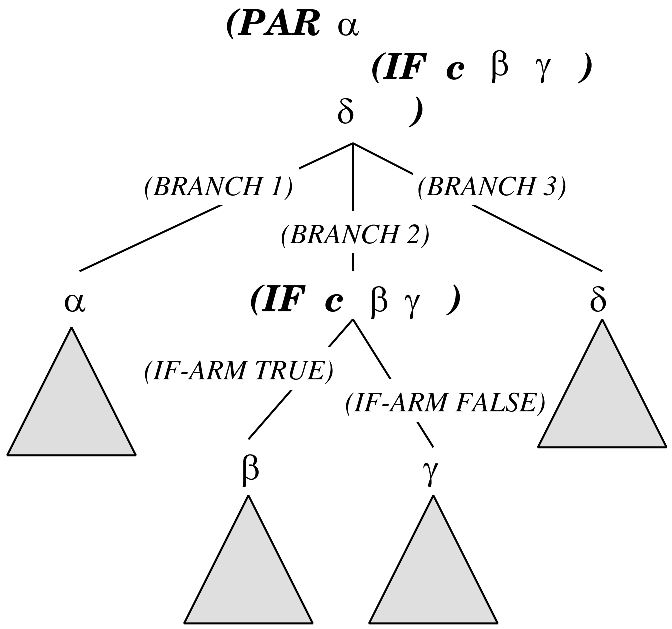


Fig. 3.1. Code tree for the RPL code piece (PAR   (IF *C* γ)).

CODE-REPLACE automatically deals with. First, the assurance of the locality of plan changes in the revision of a particular procedure call: in the case that the planner replaces code for a particular procedure call, this replacement must not a ect any other calls to the same procedure. This problem is solved by expanding procedure calls inline if the code for a particular procedure call is to be transformed.

The second problem is the automatic maintenance of references between di erent code pieces that are, for example, necessary when one piece might constrain the relative order of two other pieces. To deal with this complica-

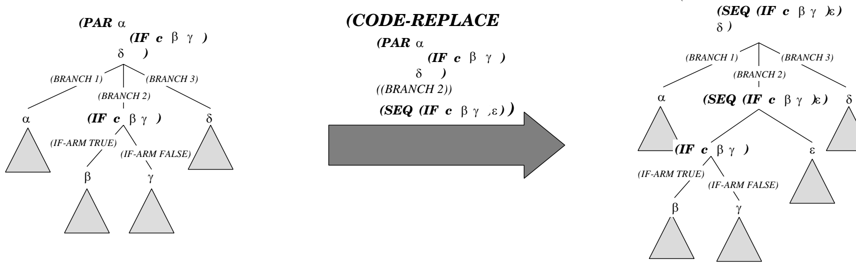


Fig. 3.2. Replacement of (IF *C* γ)) by (SEQ (IF *C* γ))   ) in the code tree representing (PLAN   (IF *C* γ)).

tion, RPL code trees allow for tagging or marking subcodes and thereby for generating unique indices for them. RPL provides a construct (:TAG *TAG C*) that gives the RPL code for *C* the symbolic name *TAG*. The value of *TAG* is the code path of the corresponding piece of the plan. As data structures, code trees contain *tag tables* that map the names into their code paths. CODE-REPLACE maintains these tag tables automatically. Thus, if the planner wants to constrain the order between a code piece   in   and   in , it replaces   by (:TAG *T*) and   by (:TAG *L*) and adds the ordering clause (:ORDER *T L*) to the top-level code. This solves our problem because whenever the plan is revised, CODE-REPLACE automatically updates the code paths for all *tagged* code pieces. Thus using tags, code pieces can reliably refer to each other, even when the structure of the code tree changes: the references to   and   are stable even if their code paths change due to transformations.

Since structured reactive controllers start planning processes that reason about parts of the controller and revise these parts if necessary, the corresponding parts of the controller have to be syntactic objects: plans. Calls to subplans are expanded because identical calls to subplans might have to be revised di erently in di erent parts of the plan. Thus, when a structured reactive controller starts a planning process for a subplan *P*, the planning process will expand subplan calls in *P* and represent the resulting plan as a code tree. If a subplan calls itself recursively, the recursive call will not be expanded.

3.1.2 RPL as a Plan Language

RPL provides several constructs that support planning by making plans more transparent and changes in plans more modular (see [124]). We have already seen the rst of these constructs, the PLAN-statement. The PLAN-statement has the form (PLAN *STEPS CONSTRAINTS*). Each *constraint* is a policy, or an (:ORDER S_1 S_2), or a (:REDUCTION *RS-OLD R-NEW* [*TAG*]). *Steps* are executed in parallel except when they are constrained otherwise. PLAN-statements are used by the planner to add ordering constraints between two subplans. Even in structured reactive plans, plan revisions that constrain the order between steps of the plan are the most common form of plan revision. As we have seen above, adding ordering constraints between two code pieces C_1 and C_2 can be done by tagging the code pieces and adding the corresponding :ORDER clause to the smallest PLAN-statement that contains both code pieces. This works for all but one case: the case in which a code piece C_1 is part of the body of a loop. We will discuss this case later in chapter 5.

Reductions are the second category of constructs that support planning. They are speci ed with the statement (REDUCE *G P [TAG]*), which tells the robot that the plan step *G* is carried out by performing the plan *P*. In this case, *G* is called the reducee and *P* the reducer. In the subsequent chapters reductions are used in two ways: rst as annotations that tell the planner what a piece of plan is supposed to do. Second, as a mechanism for recording the plan revisions that have been performed. For example, the planner reasons about subplans of the form (REDUCE *WHAT HOW*) as follows: if it needs to nd out *WHAT* the purpose of a subplan is, it tries (rst) to extract the purpose from the reducee, if the planner needs to nd out *HOW* the task is to be performed, it looks at the reducer. In order to record plan revisions REDUCE-statements are used as follows: (REDUCE *BEFORE AFTER METHOD*) where *BEFORE* is the code before and *AFTER* the code after the revision, and *METHOD* is the revision method that has been applied. This notation has two advantages. First, it makes the revisions performed by the planner transparent. Second, in order to undo a revision the planner simply reduces the REDUCE-statement to *BEFORE* again.

An important concept in classical planning is the notion of *protections* (or *causal links*) and *protection violations* (or *threads*). A protection represents an e ect of a plan step that is needed by some later step, and therefore this effect has to be maintained until the later step is executed [156]. Many classical partial-order planners work by adding ordering constraints until no protection is violated [119]. RPL provides the statement (PROTECTION *RIGIDITY PROPOSITION FLUENT REPAIR*). Protections in RPL are more expressive than those in classical planning since in RPL they do not only signal protection violations but also allow the programmer to specify the severity of a protection violation and methods for dealing with violations. An example for a protection is that the robot protects the state object in hand when delivering an object.

When to protect what, is simple to answer as long as the plans are sets of discrete steps. But consider a plan for our simulated world. A plan for the delivery of an object might contain three steps: (1) get the object, (2) go to the destination, and (3) put the object down. But when should the robot start and stop protecting the state object in hand? Ideally, the robot should start when it has the object the rst time in its hand until it intends to let it go at the destination, i.e., during get the object and put the object down. In RPL we can specify the desired behavior using the statement (SCOPE *BEGIN END BODY*) where *BEGIN* and *END* are fluents and *BODY* a piece of RPL code. *BODY* is wait-blocked until *BEGIN* becomes true and after that executed until *END* becomes true. Using the PROTECTION and the SCOPE-statement we can specify our delivery controller as follows:

```
(WITH-POLICY (SCOPE HAND-FORCE* (BEGIN-TASK LET-GO)
                    (PROTECTION :HARD (IN-HAND DESIG)
                                      (NOT (HAND-FORCE*))
                      (PICK-UP DESIG)))
   (SEQ (GET-OBJECT DESIG)
        (GO-TO DESTINATION)
        (PUT-DOWN DESIG))
```

The last two RPL constructs we will discuss in the context of planning are dynamic values and macros. Sometimes the planner has to make changes that are very low-level, like changing the routine for interpreting image data. Thus instead of adding the sensor routine as an extra argument to each perception routine it is more elegant to have a global variable SENSOR-ROUTINE* that can be dynamically bound to the sensing routine that should be used. The plan writer can then de ne a macro WITH-SENSOR-ROUTINE using the RPL command (DEF-INTERP-MACRO *(WITH-SENSING-ROUTINE ?ROUTINE . ?BODY))* and specify that in the scope of *?BODY* SENSOR-ROUTINE* is dynamically bound to *?ROUTINE.* Thus a plan revision can advice a subplan to look more carefully by doing the following reduction:

```
(REDUCE (LOOK-FOR '((CATEGORY BALL)) CAMERA-1)
        (WITH-SENSOR-ROUTINE BE-CAREFUL
            (LOOK-FOR '((CATEGORY BALL)) CAMERA-1))
        BE-MORE-CAREFUL-REVISION)
```

Tasks. Besides RPL constructs, the planner makes heavy use of the RPL concept *task*. A task is a piece of a plan *PL*, which the system intends to execute and corresponds to the execution of a node in the code tree. The interpretation of complex tasks creates subtasks. For example, executing the plan $P =$ (N-TIMES 5 (GRASP *OB*)) generates a task *T* with plan (N-TIMES 5 (GRASP *OB*)) and ve subtasks *(ITER I T)* for $I = 1,\ldots,5$ with plan (GRASP *OB*). Within the RPL interpreter this notion of task corresponds to a data structure similar to a stack frame in standard programming languages. This data structure stores the computational state of interpreting *PL*. Viewing tasks as stack frames, planning processes can extract all necessary information about the computational state of a subplan and use this information to decide what remains to be executed and how the world might look. Similarly, modifying a subplan that is currently executed requires the planner to make changes to the corresponding stack frame.

In RPL plans, tasks are rst-class objects and can therefore be passed as arguments to planning modules. In the context of RPL plans, tasks can be made explicit using the :TAG-statement and the SUBTASK-function. Executing the statement (:TAG A (GO 4 5)) statement creates a variable A and sets the value of A to the task going to location $\langle 4,5 \rangle$. Tasks can also be indexed relative to other tasks. The expression (SUBTASK *(ITER 3) T*) returns the task for the third iteration of *T* in the example above. Tasks are created on demand, i.e., when they are executed, inspected, referred to, or revised. Another function de ned on tasks is TASK->CODE-PATH, which maps any task into the code path of the RPL code that this task executes. As you might imagine, this is one of the key functions used by the planner to localize problems in an RPL plan.

3.2 The Computational Structure of Planning Processes

In chapter 2 we have viewed planning modules as black boxes that are activated (parameterized with a plan) and deactivated by the structured reactive controller. This section describes the internal structure of planning modules and their main computational processes. XFRM, the general planning framework that we use for the implementation of planning modules, provides means for building specialized planners and integrating them. This section describes a general framework, the *"criticize-revise"* cycle, in which these specialized planning algorithms can be integrated and explains afterwards how the process of anticipating and forestalling behavior flaws is embedded in it.

Let us pause for a second to prevent some possible confusion about three closely related terms: behavior flaws, plan failures, and bugs. A behavior flaw is the di erence between the predicted and intended behavior of the robot. A plan failure is a behavior flaw that is detected by a routine and signalled to the calling plans along with a failure description. A bug is a description of a behavior flaw that might contain diagnoses of the causes of the flaw and is used by the planner to index plan revisions to make the plan robust against the flaws described by the bug.

3.2.1 The Criticize-Revise Cycle

The basic control structure of a planning module is the *"criticize-revise"* cycle (see gure 3.3). The criticize step takes a plan and analyzes it in order to answer the questions how good is the plan?, what flaws in the behavior of the robot are likely to occur if the robot executes this plan?, what are the causes of these flaws?, and how much can the plan be improved by eliminating the flaws? The result of the criticize step is an *analyzed plan* that comprises the plan itself, several randomly generated execution scenarios of the plan, an average score that the plan achieved in these scenarios, and a set of bugs (for details, see chapter 6.1.1). A bug represents a flaw in the projected behavior of the robot and has a severity, an estimate of how much the plan could be improved if it were robust against flaws of this type, and a failure diagnosis. Analyzed plans are inserted into the plan queue that contains all plans generated so far, ordered according to their score.

Planning modules communicate their results and computation status through three fluents: DONE?, BETTER-PLAN?, and BEST-PLAN-SO-FAR, which are updated as follows. If in the criticize step a better plan is found, the fluent BETTER-PLAN? is pulsed and the fluent BEST-PLAN-SO-FAR set to the currently criticized plan. The fluent DONE? is pulsed when the planning e ort runs out of ways that it could try in order to improve the plan.

The second step in the criticize-revise cycle is the revise step. The revise step takes the best plan from the plan queue that has a bug. It then chooses the bug with the highest severity and applies revisions to the plan

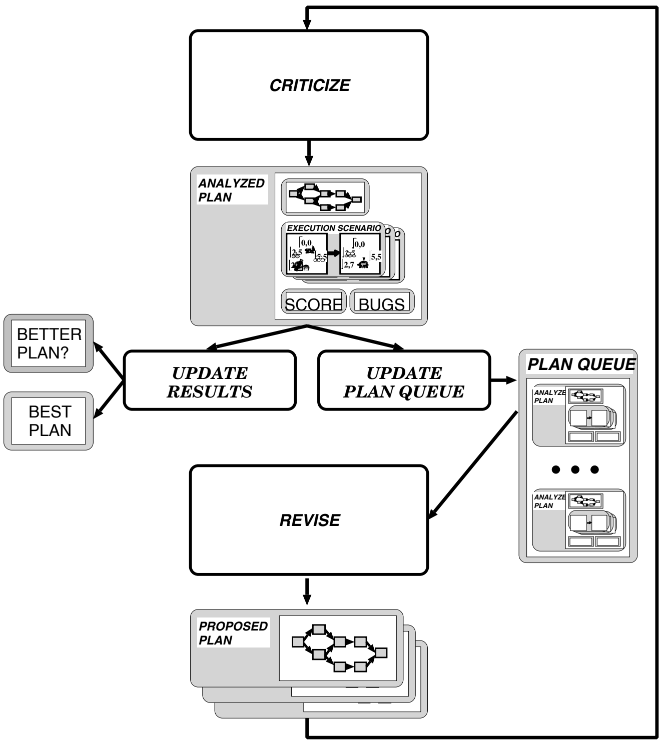


Fig. 3.3. The criticize-revise control strategy.

that try to make the plan robust against the types of flaw described by the bug. The result of the revise step is a (possibly empty) set of new plans which are passed back to the criticize step to determine whether they can be expected to improve the behavior of the robot.

Planning processes terminate when none of the plans in the plan queue contains bugs the planning process has not tried to eliminate yet, or more likely when the planning process is deactivated. The search strategy applied by planning modules is best- rst search. The nodes in the search space are plans and the root node is the plan the planning module was activated with.

The criticize-revise cycle is a variant of the generate/test/debug control strategy [151]. The generate/test/debug control strategy eliminates all bugs in a plan to obtain a correct plan. The criticize-revise control strategy tries to compute plans with optimal scores. Bugs give the planner hints about how to improve the plan. In the criticize-revise control strategy a revision that asks the robot to give up accomplishing a certain job may produce the plan with the highest score. Another di erence between the two control strategies is that the generate/test/debug control strategy tries to correct faulty planning decisions that are made in the generate phase while the criticize-revise control strategy tries to avoid behavior flaws that might be caused by routine plans when executed in non-standard situations.

3.2.2 The Criticize Step

Criticizing a plan consists of two steps: (1) Projecting several execution scenarios and (2) evaluating these scenarios with respect to the given jobs (see gure 3.4). The input for the criticize step is a structured reactive plan and the computational state of the currently executed structured reactive controller.

Plan Projection. Plan projection [127, 128] is the single most important inference technique applied by XFRM planning processes.[2] The plan projection process takes the current world model, a reactive plan, rules for randomly generating exogenous events, rules for probabilistically guessing missing pie-

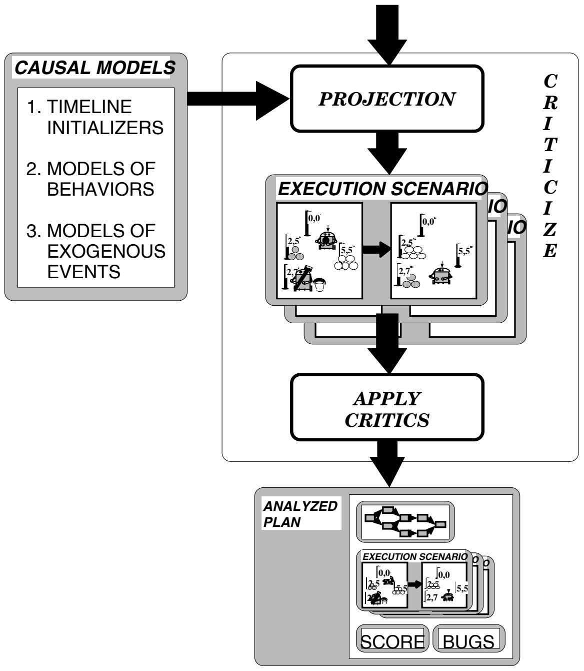


Fig. 3.4. The criticize step.

[2] The algorithms for plan projection have been developed by McDermott [127, 128]. The probabilistic temporal projection algorithm is described in [128]. McDermott [128] gives a formal semantics for the rule language introduced above, shows that a consistent set of rules has a unique model, and proves the algorithms for building and retrieving from the timeline to be correct.

ces of the world model, probabilistic causal models of basic control routines and generates *execution scenarios* of the plan.

To predict what might happen when a plan will be executed the criticize step performs a small number of Monte Carlo projections to generate possible execution scenarios. Randomly sampling a small number of execution scenarios seems to be a bad idea; so let me explain why I apply this technique and why, or better under which conditions, it is su cient for our purposes.

Since structured reactive controllers cause robots to exhibit a wide spectrum of behaviors depending on the speci c circumstances in which they are executed, the planning processes can neither compute bounds for di erent plan outcomes [91] nor their exact probabilities [107, 103].

Behavior modules robot cause temporally complex behavior.[3] They cause changes in the world when they start, terminate, and change states while they are active. For example, the process of going to a particular locations causes the robot to change its location while the process is active. In addition, the e ects caused by concurrent processes often interfere and therefore processes when run together with other processes cause e ects that are very di erent from the ones they cause when run in isolation. Consider a plan that asks the robot to go to some location and look for an empty box whenever the camera is not used. The e ects of looking for empty boxes might be very di erent depending on when the routine gets activated (because the robot moves). Another factor that increases the number of possible execution scenarios drastically is that RPL contains non-deterministic control structures. For this reason many structured reactive controllers are non-deterministic with respect to which subplans they execute, when, and for how long.

Besides the rich temporal structure of the behaviors generated by structured reactive controllers, incomplete and uncertain information increases the number of possible execution scenarios even more. The duration of the processes run by behavior modules cannot be predicted accurately and the processes often have probabilistic e ects. In addition, the interpretation of the controllers is driven by the feedback from control processes and sensor readings possibly caused by random exogenous events that may occur at any time.

An alternative might seem to enumerate execution scenarios according to their likelihood. But this technique breaks down even in very simple scenarios. Consider a biased dice: the probability that we roll a one is fty percent, while the probability of rolling all other numbers is ten percent. Now, the robot has to project the plan roll the dice until it shows a six to approximate the likelihood that the plan will roll a six. If projector is biased to look at the most likely outcomes rst, the plan will be predicted never to succeed while in reality the plan will certainly succeed.

[3] Allen and Ferguson [6] discuss important issues in the representation of temporally complex of actions and events.

The only existing temporal projection algorithms that can project plans that cause probabilistic and such temporally complex behavior are Monte Carlo projection algorithms [128, 165].

This discussion immediately raises a second question: how can sampling a small number of possible execution scenarios su ce to anticipate and forestall behavior flaws? The reason is that structured reactive controllers contain routine plans. Routine plans are stable in standard situations because they monitor the feedback of the control processes and sensor readings and locally recover from many execution failures. Consider, for instance, a random event that causes objects to slip out of the robot's hand. Such events might occur at any time during deliveries and, as a consequence, the object might fall down at many di erent locations. However, the delivery routine has a policy that speci es whenever the handforce sensor drops to zero, interrupt the delivery, pick up the object again, and then continue with the delivery. Thus whether and where the object falls down does not matter very much because the robot will immediately pick it up again: when executing the routine plan the overall behavior of the robot is fairly independent of objects slipping out of the robot's hand.

The conditions under which a small number of projected execution scenarios su ce to improve the average performance of a robot by anticipating and forestalling behavior flaws are that

1. it is unlikely that routine plans cause behavior flaws in standard situations; and
2. if a routine plan is likely to cause a behavior flaw in situations that satisfy a condition c and the robot has detected c then the behavior flaws will probably be projected.

Later, in chapter 8.2.2 we will evaluate the conditions under which a small number of Monte Carlo projections are su cient more carefully.

Execution scenario. An execution scenario represents how the execution of a robot controller might go, that is, how the computational state and the environment changes as the structured reactive controller gets executed. An *execution scenario* consists of a *timeline* and a *plan interpretation trace*, and temporal, causal, and teleological relationships between plan interpretation, the world, and the physical behavior of the robot. A *timeline (chronicle)* represents how the world changes and a *plan interpretation trace* represents how the computational state of the reactive plan changes as the plan gets executed. When the plan is executed, the plan interpreter generates a structure of stack frames called the *task network*. A task corresponds to the execution of a plan piece. When the interpreter is run in projection mode, all these stack frames are preserved for later inspection by plan transformations.

The timeline is a linear sequence of events and their results, synchronized with the task network. Timelines represent how the world changes during the execution of a plan in terms of *time instants*, *occasions*, and *events* [128]. *Time* *instants* are points in time at which the world changes due to an action of the robot or an exogenous event. Each time instant has a *date* which holds the time on the global clock at that particular time instant occurred. An *occasion* is a stretch of time over which a world state P holds and is speci ed by a proposition, which describes P, and the time interval for which proposition is true. More than one event can occur at a time instant, but only one of them can e ect changes in the world. We will call this the *causing event* and the set of all occurring events the *happenings* of the time instant.

Guessing the current world state. The rst step in plan projection is guessing the current state of the environment. Three sources of evidence help the projector to do this.

1. Perceptions taken by the robot so far, in particular the designators stored in the variable KNOWN-THINGS. These perceptions include the non-standard situations that have been detected by the robot such as other robots and unexpected objects that might get confused with those for which the robot's jobs are to be accomplished.
2. A priori knowledge about how the environment typically looks. This knowledge comprises how many objects are typically spread out in the world, where, what kinds of objects, and which colors the objects have. A priori knowledge is stored in form of probability distributions. Although there are typically other robots in its environment the prior knowledge says there are none. Ignoring other robots in the prior knowledge is the right thing to do because it makes little sense to make a plan robust against dreamed up robots. Thus the only robots in the current state are those that the robot has seen and observed the activities of.
3. Commands given by the supervisor. If the robot gets a command to deliver all balls from location S to location D it makes sense to assume that there are two or three balls at S and to evaluate the routine plan under this assumption. If it turns out later that there are twenty balls at S then the robot controller can cancel the current planning e ort and restart a new one which starts out with twenty balls in the beginning.

You might think that there is a lot of guessing going on and that the situation guessed by the projector might not have enough in common with the current state of the environment for the results of the planning e orts to be relevant. However, if you think about it the planner does not have to guess the current situation exactly because it can expect its routine plans to work in all standard situations. The only requirement we need is that the guessed state is atypical in the same ways as the current situation. And this holds for our projector because the robot monitors its surroundings for atypical aspects of the environments, stores detected atypical aspects in global variables, and the projector guesses the initial state for the projections based on values of these variables.

The process of guessing the current situation is interesting for two other reasons. First, this is the step where the data structures used for controlling the robot are translated into symbolic representations. Second, it is computationally prohibitively expensive to guess the complete state of the environment and to predict the evolution of the whole environment. If you consider our world simulator, for a given set of commands only a very small part of the environment is relevant. Therefore, our projector performs lazy temporal projection. It projects whatever is necessary to decide what events occur. But it projects how the environment changes only if these changes have to be known to predict the subsequent events (cf. [90]).

Timeline initializers are pattern-directed procedures that guess the current situation of the environment. If there is a query about the state in which the plan projection starts then timeline initializers rst look up whether the global variables contain information concerning the query. If so, the timeline initializer guesses the answer to the query based on this information, the probabilistic model of the sensor operation that provided the information and the probability that the information might have become outdated. If the robot has not made any perceptions related to the query then the timeline initializer has to guess the answer to the query based on the robots prior probabilistic models. The answers to these queries are then asserted as propositions to the initial state of the timeline.

Projecting plan execution. Now that we know how the projector constructs the initial state of the timeline we can turn to the question of how to project the execution of a structured reactive plan. The answer is quite simple: the projector does the same thing the interpreter would do, except that whenever the executed robot controller interacts with the real world, the projected robot controller interacts with the timeline. The places where this happens are the low-level plans (see chapter 4.3.2). Thus instead of executing a low-level plan the projector guesses the results of executing these plans and asserts their e ects in form of propositions to the timeline. Since the main purpose of low-level plans is the activation and deactivation of behavior modules, let us recall the most salient features of behavior modules. Behavior modules can be activated and deactivated; they signal their successful (FINISH *VALUES*) or unsuccessful (FAIL *DESCRIP*) completion. While running they cause changes in the computational state of the robot controller (by setting fluents) and changes in the environment and the physical state of the robot.

The projector uses *projection rules*, rules that describe the behavior of low-level plans, as models of what the plans do.

```
(PROJECT (NAME    ARGS   )
    COND
    ( $D_1$ $EV_1$ ... $D_n$ $EV_n$ )
    OUTCOME  )
```

Thus, instead of activating the behavior module *NAME* the projector res the projection rule. If a projection rule is red it rst checks whether *COND* is satis ed in the last timeinstant *TI* of the timeline. If so it appends the events EV_j at the time $TI + \quad _{1 \ j \ n} d_j$ to the timeline unless they have the form (LISP *EXP*). If they have the form (LISP *EXP*) they describe changes in the computational state of the controller rather than changes in the world, i.e., these events set global variables and fluents of the controller. The *OUTCOME* is either of the form (FINISH *VALUES*) or (FAIL *DESCRIP*). This statement speci es the signal that is sent by the behavior module when the behavior is completed. This signal is caught and handled by the interpreter in projection mode.

Typically, in order to describe a behavior module we need a set of projection rules that probabilistically describe the possible event sequences and outcomes when activating a behavior module. In each situation exactly one projection rule is applied (although the decision about which one it is might be probabilistic).

The gure 3.5 shows a projection rule that models the normal behavior that the robot exhibits when running the behavior module LOOK-FOR. The condition of the projection rule determines where the robot is *(1)*, probabilistically decides whether the LOOK-FOR is normal based on the camera used and the speci cs of the location *(2)*, reconstructs what objects are located there *(3)*, and then uses the sensor model for the camera in order to probabilistically decide how the robot perceives each object. For each object that is perceived as matching the perceptual description ?PL a local designator ?DESIG is created and collected in the variable DESIGS. The last condition *(7)* estimates the time ?DT after which the LOOK-FOR behavior completes. Upon completion of the LOOK-FOR behavior the projected interpretation process

```
(PROJECTION (LOOK-FOR ?PL ?CAM)
    (AND (LOC ROBOT (COORDS ?X ?Y))                            (1)
         (NORMAL-LOOK-FOR-BEHAVIOR ?CAM ?LOC)                  (2)
         (SETOF ?OB (LOC ?OB (COORDS ?X ?Y)) ?OBS-HERE)        (3)
         (SENSOR-MODEL ?CAM ?SENSOR-MODEL)                     (4)
         (FEATURES ?PL ?FEATURES)                              (5)
         (SETOF ?DESIG)                                        (6)
                (AND (MEMBER ?OB ?OBS-HERE)
                     (OBJ-SEEN ?OB ?SENSOR-MODEL)
                     (PERCEIVED-PROPERTIES
                        ?OB ?FEATURES ?SENSOR-MODEL ?PL)
                     (LOCAL-DESIG ?DESIG ?OB ?PL ?X ?Y))
                ?DESIGS)
         (LOOK-TIME (COORDS ?X ?Y) ?FEATURES ?DT))             (7)
    (0   (BEGIN (LOOK-FOR ?PL ?CAM))
     ?DT (END (LOOK-FOR ?PL ?CAM))
     0   (LISP (PULSE (VISUAL-INPUTS-FLUENT ?CAM))
               (SET-OBS-POS-FLUENT ?SEEN ?CAM)))
    (FINISH ?DESIGS))
```

Fig. 3.5. A projection rule describing the behavior module LOOK-FOR.

sends a success signal with the return value ?DESIGS as its argument. The projected behavior consists of three events: two that change the world (BEGIN (LOOK-FOR ?PL ?CAM)) and and (END (LOOK-FOR ?PL ?CAM)), which occurs ?DT later. The third event changes the computational state of the structured reactive controller after passing ?DT time units. This event pulses the fluent (VISUAL-INPUTS-FLUENT ?CAM) and sets the fluent (OBS-POS-FLUENT ?CAM).

Besides asserting the events that take place during the execution of a plan we have to specify how these events change the world. This is done using *e ect rules*. E ect rules have the form (E->P-RULE *COND EV PROB EFFECTS*) and specify that an event *EV* under condition *COND* has with probability *PROB* the e ects *EFFECTS*. *EFFECTS* are either propositions which speci es that *EV* causes the corresponding proposition to be true or have the form (CLIP *PROPS*) which speci es the *EV* causes *PROP* not to hold any longer.

How the event (END (LOOK-FOR ?PL ?CAM)) changes the environment is projected by *E->P* rules. One of them is shown in gure 3.6. The rule speci es that if at a time instant at which an event (END (LOOK-FOR* ?PL ?CAM)) occurs the state (VISUAL-TRACK ?DESIG ?OB) holds for some ?DESIG and ?OB, then the states (VISUAL-TRACK ?DESIG ?OB) will not (with a probability of 1.0) persist after the event has occurred, i.e., they are clipped by the event.

The projector also allows for modeling exogenous events. Exogenous event models are represented by rules of the form (P->E *COND DUR EV*) where *DUR* is the average time that *EV* occurs under condition *COND*. Thus each time before the projector asserts a new event   later from the last event it asks whether *COND* holds and if so randomly decides whether an event of type *EV* has occurred within . If so it randomly generates a time   between 0 and   at which *EV* is projected to occur.

An Example of a Projected Execution Scenario. Let us now look at the following small RPL plan and one of its projected execution scenarios. For simplicity reasons the plan does not apply complex control structures to be more robust. The plan contains a subplan tagged COMMAND-2 that asks the robot to go to location ⟨*2,2*⟩, look for a ball, and track it. If no ball is found, the plan will fail. The designators that are returned by the LOOK-FOR-routine are stored in the local variable OBS. The plan tagged COMMAND-2 is a subplan of the RPL-statement TOP-LEVEL that speci es that the subplan is to be treated as a user command, that is a report about the success

```
(E->P (VISUAL-TRACK ?DESIG ?OB)
      (END (LOOK-FOR* ?PL ?CAM))
      1.0
      (CLIP (VISUAL-TRACK ?DESIG ?OB))
```

Fig. 3.6. An E->P-rule describing the e ects of the event (END (LOOK-FOR ?PL ?CAM)).

or failure of the subplan is signalled upon the completion of the plan. The WITH-POLICY-statement speci es that the plan is constrained by the policy (CHECK-SIGNPOSTS-WHEN-NECESSARY). This policy asks the robot to stop and look for the signpost at the robot's current location. It is activated whenever a subplan signals that it needs to know the robot's current location and the robot has moved since the last time it looked for a signpost.

```
(WITH-POLICY (CHECK-SIGNPOSTS-WHEN-NECESSARY)
    (PARTIAL-ORDER
        ((:TAG MAIN
                (TOP-LEVEL
                    (:TAG COMMAND-2
                        (SEQ (GO 2 2)
                            (LET ((OBS ()))
                                (!= OBS (LOOK-FOR
                                            '((CATEGORY BALL)) 0))
                                (IF (NOT (NULL OBS))
                                    (LET ((OB (CAR OBS)))
                                        (START-TRACKING OB 0))
                                (FAIL))))))))))))
```

We project this plan for an initial situation, in which the robot starts at location ⟨*0,0*⟩ and two balls are placed at ⟨*2,2*⟩ (see gure 3.7). The tasks that are created during the projection are stored in the task network in the upper part of gure 3.7. This enables the planner to infer which subplans were executed and caused the execution scenario.

When the plan is projected the routines (CHECK-SIGNPOSTS-WHEN-NECESSARY) and the PARTIAL-ORDER-subplan start in parallel. The policy is waitblocked until some subplan requests to know the current location. The interpretation of the PARTIAL-ORDER proceeds until the (GO 2 2) starts. To go to location ⟨*2,2*⟩ the routine has to know where the robot is right now and therefore issues a request. Upon signalling the request the policy (CHECK-SIGNPOSTS-WHEN-NECESSARY) wakes up.

Instead of controlling the real camera to receive sensor data, the projector asserts four events to the timeline: the begin and end of looking for an object of the category signpost and the begin and end of examining the signpost. These events are speci ed by the projection rules for the low-level plan LOOK-FOR-SIGNPOST. The e ects of these events are speci ed by the E->P rules for LOOK-FOR-SIGNPOST. The application of these causal rules asserts occasions to the timeline (see gure 3.8). In our case, the projector predicts that the robot will see SIGNPOST14 and recognize it as a signpost as a result of the event (END (LOOK-FOR-SIGNPOST)).

After knowing its location the robot continues the execution of the GO-subplan, which is recorded by the projector by asserting the event (BEGIN (GO 2 2)) to the timeline. To project the behavior of the robot that is caused by the GO-process the projector probabilistically estimates the time it takes the robot to get to location ⟨*2,2*⟩. Based on this estimated duration the projector asserts an event (END (GO 2 2)) at the timeinstant with time stamp

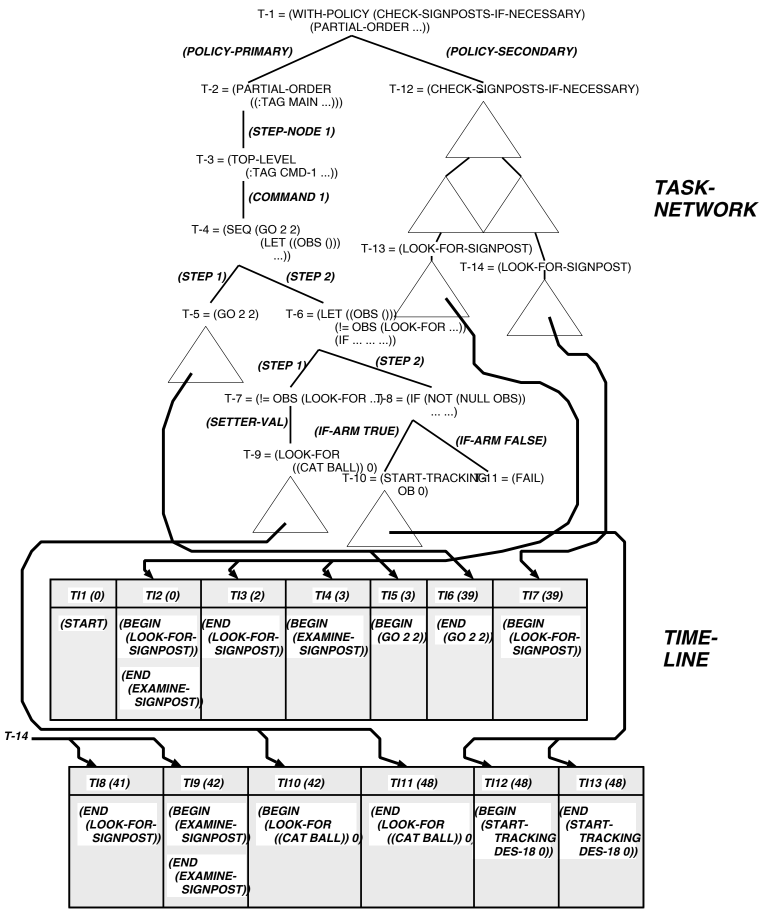

T-1 = (WITH-POLICY (CHECK-SIGNPOSTS-IF-NECESSARY) (PARTIAL-ORDER ...))
(POLICY-PRIMARY)
(POLICY-SECONDARY)
T-2 = (PARTIAL-ORDER ((:TAG MAIN ...)))
T-12 = (CHECK-SIGNPOSTS-IF-NECESSARY)
(STEP-NODE 1)
T-3 = (TOP-LEVEL (:TAG CMD-1 ...))
TASK-NETWORK
(COMMAND 1)
T-4 = (SEQ (GO 2 2) (LET ((OBS ())) ...))
T-13 = (LOOK-FOR-SIGNPOST)
T-14 = (LOOK-FOR-SIGNPOST)
(STEP 1)
(STEP 2)
T-5 = (GO 2 2)
T-6 = (LET ((OBS ())) (!= OBS (LOOK-FOR ...)) (IF ... ...))
(STEP 1)
(STEP 2)
T-7 = (!= OBS (LOOK-FOR ...))
T-8 = (IF (NOT (NULL OBS)) ... ...)
(SETTER-VAL)
(IF-ARM TRUE)
(IF-ARM FALSE)
T-9 = (LOOK-FOR ((CAT BALL)) 0)
T-10 = (START-TRACKING OB 0)
T-11 = (FAIL)
TI1 (0)
TI2 (0)
TI3 (2)
TI4 (3)
TI5 (3)
TI6 (39)
TI7 (39)
(START)
(BEGIN (LOOK-FOR-SIGNPOST))
(END (EXAMINE-SIGNPOST))
(END (LOOK-FOR-SIGNPOST))
(BEGIN (EXAMINE-SIGNPOST))
(BEGIN (GO 2 2))
(END (GO 2 2))
(BEGIN (LOOK-FOR-SIGNPOST))
TIME-LINE
T-14
TI8 (41)
TI9 (42)
TI10 (42)
TI11 (48)
TI12 (48)
TI13 (48)
(END (LOOK-FOR-SIGNPOST))
(BEGIN (EXAMINE-SIGNPOST))
(END (EXAMINE-SIGNPOST))
(BEGIN (LOOK-FOR ((CAT BALL)) 0)
(END (LOOK-FOR ((CAT BALL)) 0)
(BEGIN (START-TRACKING DES-18 0))
(END (START-TRACKING DES-18 0))

Fig. 3.7. Task network and the timeline that are generated by projecting the plan for COMMAND-2. The timeline is a sequence of timeinstants, where *TI1 (0)* means the timeinstant *TI1* has occurred at time *0*. The entries for each time instant list the events that caused the time instant. Further the tasks in the task network point at the events that they cause.

39. In addition, it asserts an occasion (TRAVELLING ROBOT *0,0* *2,2*) but no location assertions for the robot: going from ⟨*0,0*⟩ to ⟨*2,2*⟩ is a continuous process and in most of the cases the planning process does not have to know where the robot is while going to ⟨*2,2*⟩. Thus the representation of the the go behavior is very simple: retracting the current position of the robot upon activating GO and asserting the new position of the robot after the go behavior is completed.

The simple representation of the GO-behavior su ces as long as nothing important happens while going to ⟨*2,2*⟩. But suppose that an exogenous event that causes an object to slip out of the robot's hand is projected while the

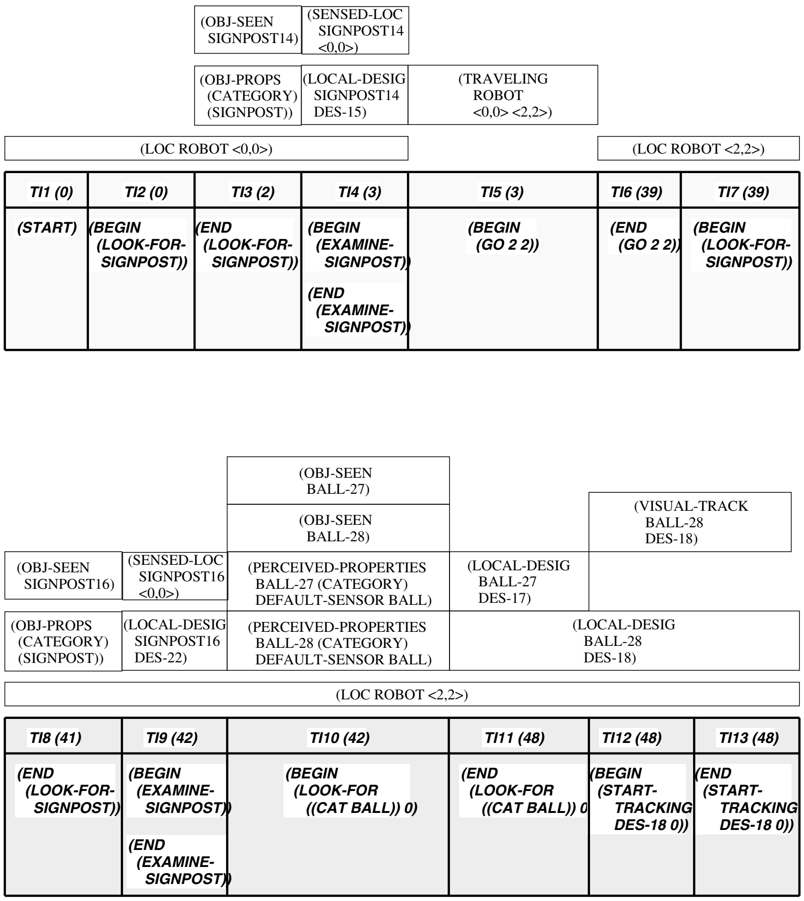

(OBJ-SEEN SIGNPOST14)
(SENSED-LOC SIGNPOST14 <0,0>)
(OBJ-PROPS (CATEGORY) (SIGNPOST))
(LOCAL-DESIG SIGNPOST14 DES-15)
(TRAVELING ROBOT <0,0> <2,2>)
(LOC ROBOT <0,0>)
(LOC ROBOT <2,2>)
TI1 (0)
TI2 (0)
TI3 (2)
TI4 (3)
TI5 (3)
TI6 (39)
TI7 (39)
(START)
(BEGIN (LOOK-FOR-SIGNPOST))
(END (LOOK-FOR-SIGNPOST))
(BEGIN (EXAMINE-SIGNPOST))
(END (EXAMINE-SIGNPOST))
(BEGIN (GO 2 2))
(END (GO 2 2))
(BEGIN (LOOK-FOR-SIGNPOST))
(OBJ-SEEN BALL-27)
(OBJ-SEEN BALL-28)
(VISUAL-TRACK BALL-28 DES-18)
(OBJ-SEEN SIGNPOST16)
(SENSED-LOC SIGNPOST16 <0,0>)
(PERCEIVED-PROPERTIES BALL-27 (CATEGORY) DEFAULT-SENSOR BALL)
(LOCAL-DESIG BALL-27 DES-17)
(OBJ-PROPS (CATEGORY) (SIGNPOST))
(LOCAL-DESIG SIGNPOST16 DES-22)
(PERCEIVED-PROPERTIES BALL-28 (CATEGORY) DEFAULT-SENSOR BALL)
(LOCAL-DESIG BALL-28 DES-18)
(LOC ROBOT <2,2>)
TI8 (41)
TI9 (42)
TI10 (42)
TI11 (48)
TI12 (48)
TI13 (48)
(END (LOOK-FOR-SIGNPOST))
(BEGIN (EXAMINE-SIGNPOST))
(END (EXAMINE-SIGNPOST))
(BEGIN (LOOK-FOR ((CAT BALL)) 0)
(END (LOOK-FOR ((CAT BALL)) 0
(BEGIN (START-TRACKING DES-18 0))
(END (START-TRACKING DES-18 0))

Fig. 3.8. The gure shows the timeline in more detail. It shows the timeinstants and the occasions, the states that are true and their duration. Several predicates (VISUAL-TRACK, LOCAL-DESIG) represent relations between objects in the world and computational objects such as designators.

robot is going. To predict the new location of the object the projector has to predict the location of the robot at the time of the event. This is done as follows: the projector estimates the location l' of the robot at the time t' when the event occurs. It asserts the location of the robot at time t' to be location l' and substitutes the state going from l to d by two states going from l to l' and going from l' to d. Thus continuous events are assumed to be discrete as long as no state a ected by the continuous event is queried for a time at which the event occurs. If a state a ected by a continuous event needs to be inferred, the intermediate state s is estimated and the continuous event replaced by two events one from the initial state to the intermediate one and one from the intermediate state to the nal one.

```
TASK T-6
   SUPERTASK           T-4
   TASK-NAME-PREFIX    (STEP 2)
   TASK-CODE-PATH      ((POLICY-PRIMARY) (STEP-NODE 1)
                        (COMMAND 1) (STEP 2))
   TASK-ENVIRONMENT OBS   (BEGIN-TASK T-4)   ()
                          (END-TASK T-7)     (DES-17 DES-18)
   TASK-STATUS         TI10  CREATED
                       TI10  ACTIVE
                       TI13  DONE
```

Fig. 3.9. Conceptual view of a projected task. The task environment contains the variables in the scope of the task, their values at di erent stages of execution, and state of plan interpretation when these values were assigned. The task status contains the change of the task status during the projected plan interpretation and when the status changed.

But let us continue with the description of the projection of the example plan. Upon arrival the GO-subplan checks whether it has arrived at the right location by signalling (CHECK-SIGNPOSTS-WHEN-NECESSARY) to look for the signpost which yields events that cause the timeinstants TI7, TI8, and TI9. The next step is to look for the balls. The (END (LOOK-FOR ((CAT BALL)) 0)) event causes the states (LOCAL-DESIG DES-17 BALL-27) and (LOCAL-DESIG DES-18 BALL-28), which represent that the designator DES-17 was returned by the LOOK-FOR routine when the routine detected BALL-27. Next, the (BEGIN (START-TRACKING DES-18)) event asserts the occasion (VISUAL-TRACK DES-18 BALL-28), which represents that the robot tracks DES-18 by pointing the camera at BALL-28.

Besides changing the world, the execution changes the computational state of the structured reactive controller, which is recorded in the task network. For example, consider the task T-6 that is shown in gure 3.9. In the projected execution scenario the task is created and activated at timeinstant T10 and is successfully completed (its task status is DONE) at timeinstant T13. The task environment of T-6 contains the local variable OBS that is initialized with the empty list (). Later at the end of task T7 the value of OBS is set to the list containing the designators DES-17 and DES-18.

The Application of Critics. A planner implemented according to the criticize-revise paradigm cannot do anything unless it has *critics* that nd flaws and *eliminators* that produce plans that do not have these flaws. A *critic* is a function that can be applied to execution scenarios to determine flaws in the projected robot behavior. Examples of such critics are the protection violation critic, the deadline violation critic, the unscheduled plan critic, and the perceptual confusion critic. The protection violation critic identi es events in an execution scenarios that cause a protected state to be clobbered and describe the violation in form of a *protection violation* bug. The deadline violation critic checks for each top-level command with a speci ed deadline, whether the plan for the top-level command succeeded before the deadline passed. The unscheduled plan critic estimates how much a plan can be improved by scheduling plan steps such that the travel time of the robot is minimized. The perceptual confusion critic identi es situations in the projected execution scenarios in which the robot manipulated the wrong objects because of using ambiguous designators. Critics represent each flaw in the robot behavior that they have found as a bug and estimate how much the plan could be improved by eliminating the flaw.

3.2.3 The Revise Step

With each class of bugs there is associated a function that takes a bug and a plan and produces a set of plans. Those revision methods use the bug description and revise the plans to get rid of flaws of this kind.

3.3 The XFRM Planning Framework

The criticize-revise paradigm is implemented as the XFRM planning framework. In some ways, XFRM is like a tool for building expert systems: it is an empty shell for building transformational planners for robot controllers written in RPL. XFRM provides several powerful and general tools for transformational planning of reactive behavior. We have already seen the rst of these tools: the temporal projector together with the language for representing the e ects of behavior modules. The temporal projector is a key tool in a planning system because the prediction of what might happen when a plan gets executed forms the basis for the decisions about how to revise plans.

In addition, XFRM comes with a library of specialized planners which the programmer may or may not use to build a planner for a particular robot and methods for integrating specialized planners into a coherent planning module. Additional special-purpose planners can be implemented and plugged into the system as needed. A specialized planner searches for particular types of flaws and revises the plan such that the occurrence of these types of flaws are less likely. To embed a new specialized planner into the XFRM framework, a programmer has to do three things: (1) de ne a class of bugs, (2) de ne a *critic* that given a projected execution scenario, nds all occurrences of bugs of this class, and (3) de ne an *eliminator* that can, given a plan and a bug produce a set of candidate plans that are supposed to be better than the original plan. Examples of specialized planners are schedulers and eliminators for blown deadlines, perceptual confusions, and behavior flaws caused by overloading the robot.

A bug class has class-speci c slots in which instances of the class store information about detected flaws and explanations for their occurrence. Beside the class speci c slots, each bug class must have two methods: one for determining whether two flaws in the projected behavior are instances of the same bug and another one for estimating the severity of a bug. The critic is a function that takes an execution scenario and computes all occurrences of bugs of a particular bug class and summarizes them with a bug description. The bug description provides the information required by the corresponding bug eliminator to produce new candidate plans.

3.4 Anticipation and Forestalling of Behavior Flaws

The anticipation and forestalling of flaws is a very general class of planning operations. Classical planning relies entirely on this kind of planning operations. Classical planners anticipate and forestall two kinds of flaws: the ones caused by missing plan steps and the ones caused by negative interferences between plan steps. This is su cient because classical planning makes strong assumptions. These assumptions include that the objects in the robot's environment have unique names that are su cient for the robot to manipulate them, that the agent is omniscient, has perfect models of its actions, and that all changes in the environment result from the agent's own actions. Making these assumptions reduces the number of ways in which plans can fail: Assuming that objects are known and can be manipulated using their names allows the planner to neglect di cult problems in robotic sensing. Assuming omniscience excludes all failures caused by incorrect and incomplete world models. If we are willing to make strong enough assumptions as in classical planning we can devise small and elegant planning algorithms that compute provably correct plans.

For a planner that is embedded in an autonomous robot, however, the anticipation and forestalling of execution failures is both more general and more di cult. Embedded planners are often forced to give up assumptions that are unrealistic. Giving up assumptions usually means for a planner that it has to deal with additional kinds of behavior flaws. A planner that does not make the assumption that the robot can refer to objects in the world by their names will need a more detailed model of how the robot perceives objects and uses its perceptions to manipulate objects. If this model describes perception as imperfect and incomplete, the planner might have to consider that a plan might fail because the robot overlooks objects or cannot correctly perceive an object's visual properties. Also, if the robot perceives only what it needs to perform its tasks or the visual properties of objects are not unique then plans might fail because of ambiguous object descriptions, and so on. As we drop assumptions about the robot, its jobs, and its environment, robot planners have to deal with more and more kinds of behavior flaws.

Forestalling behavior flaws is implemented as a specialized planner in the XFRM framework [21]. The computational process of anticipating and forestalling behavior flaws executes the following sequence of computation steps: (1) predict the consequences of executing a plan; (2) detect behavior flaws in the projected execution scenarios; (3) diagnose the reasons for the behavior flaws; and (4) revise the plans in order to eliminate the failures.

3.4.1 The Detection of Behavior Flaws

What are behavior flaws and how can they be detected? Given a projected execution scenario, the specialized planner for forestalling execution failures has to decide whether the top-level commands have been achieved. The top-level commands given by the user specify constraints on the behavior of the robot and the way the world is supposed to change. Sometimes these constraints are not met by the projected execution scenarios. We say in this case that a *plan failure* occurred. A plan failure is then simply a violation of these constraints. For example, the command *"get all the balls from location ⟨0,8⟩ to the location ⟨0,10⟩"* is a constraint that requires that each block that has been at location *⟨0,8⟩* will be at location *⟨0,10⟩* when the robot is done. An execution failure describes a ball that has been at *⟨0,8⟩* but is not at *⟨0,10⟩* when the robot is done.

3.4.2 Behavior Flaws and Plan Revisions

Whether a particular plan revision is appropriate depends on the type of a flaw and the information the planner is able to extract from a projected execution scenario. There are two general observations that are worth making. First, plan revisions tend to be better the more information they use and the more specialized they are. Second, there are generally useful plan revisions that require very little information about the flaw and the structure of the plan. The second observation is important in the context of planning reactive behavior because sometimes parts of reactive plans are opaque to the planner, or the plan interpreter requires revisions before failure analysis is completed.

These two characteristics of revising structured reactive plans can be illustrated using the *Sussmann anomaly* [155]. So let us leave the DeliveryWorld for a moment and consider the following problem in the blocks world. Suppose we give the conjunctive goal (AND (ON A B) (ON B C)) to the planner and in the current state C is on top of A and B is on the table. The default plan for (ACHIEVE (ON ?X ?Y)) looks as follows: remove everything on top of ?X and ?Y and then put ?X on ?Y. At the end: look whether ?X is on ?Y. The default plan for a conjunction of goals is to achieve the subgoals in a random order. Giving the conjunctive goal to the planner would lead to a behavior flaw: no matter in which sequence the robot accomplishes its subgoals, it would not never achieve a state in which both subgoals hold. Even worse, the robot would not recognize that the plan failed because it only checks whether the individual goals have been achieved.

If the robot planner does not know anything about what went wrong, it can ask the plan interpreter to perform the task repeatedly until it is carried out successfully by wrapping the control structure (N-TIMES <*n*> UNTIL <*goal*>) around the conjunctive goal. If the plan tries the conjunctive goal at least three times, the plan would not fail because of interactions between the two subgoals. Trying plan steps repeatedly and monitoring the success is one of the basic methods used in RAP to make the reactive plans robust [70]. Of course, this plan revision method is a heuristic. It cannot improve the plan if the world is (semi-)Markovian and the flaw is caused by some hidden state. The planning process will notice this when it projects the plan that results from the application of the plan revision methods. The resulting plans would be judged inferior to the original plan because they cause the same behavior flaws and waste time resources by multiply and unsuccessfully trying to cause the desired behavior.

An alternative revision is to introduce an additional subgoal (ON C TABLE) and add ordering constraints on the subgoals. The revised plan requires the minimal number of physical actions to achieve the task. However, for this revision to apply, the planner has to know that the a goal was clobbered by the robot's own action. Knowing the behavior flaw was caused by a task interference, the planner can apply a goal achievement procedure of a partial-order planning system (e.g. [119]) to resolve the problem.

Whenever possible, the planner should apply the most speci c or informed plan revisions. The second revision for the *Sussman anomaly* flaw is better because the revision deals with flaws due to task interferences instead of flaws in general.

The plan revisions for the Sussman anomaly are typical. Often plan revisions are quite coarse and might not require a detailed understanding of the causes of a behavior flaw or subplan manipulated. Consider, for instance, the case where an external event destroys an already achieved goal. A coarse-grained revision could wrap a reaction around the existing piece of code that monitors the goal and reachieves it whenever it becomes false. Plan revisions can be so coarse because RPL is so expressive and provides the planner with powerful control structures like trying plans repeatedly until they succeed, or monitoring the execution of a subplan and running error recovery strategies triggered by sensory information. Since the scope of monitoring can be chosen such that the planner is on the safe side and since control structures can recover from di erent kinds of failures, behavior flaws do not have to be narrowed down or explained perfectly.

3.4.3 The Diagnosis of Behavior Flaws

The planner starts the revision of robot plans with an understanding of the relevant parts of the plan and tries to explain what happened in the execution scenario in terms of the planner's model of the plan and the world. Let us illustrate how this works by continuing with our example from above and considering the case that in a given execution scenario the command *"get all the balls from location ⟨0,8⟩ to the location ⟨0,10⟩"* has not been completely accomplished. If a ball is not at its destination at the end of an execution scenario a good rst diagnosis step is to check whether the ball has been at the destination at some time. If not the planner can try to nd out why it has not been delivered. There are two kinds of scenarios in which balls have not been delivered. The rst ones are those in which the robot tried to deliver a ball, but was not successful. In the second sort of scenarios the robot did not even try to deliver the ball. If it did not try to deliver the ball, the robot might have overlooked the ball, or not recognized it as a ball, or the ball might have disappeared while the robot was carrying out its plan. Thus, a good strategy is narrowing down the causes of behavior flaws to more and more speci c types of flaws.

We de ne a taxonomy of models of behavior flaws where each model describes a particular class of behavior flaws. A classi cation algorithm that takes a flaw description and computes the set of most specialized re nements with respect to the given flaw taxonomy. With each class of behavior flaws there are associated (1) plan revisions to make the plan more robust if flaws of this category occur and (2) conditions re ning and narrowing down the observed behavior flaw.

The taxonomy in gure 3.10 shows the categories of behavior flaws that can occur in the performance of achievement tasks when they are carried out using *imperfect sensors*, *imperfect control*, an *inaccurate and highly incomplete world model*, and *with other tasks*. The taxonomy is used for the analysis of behavior flaws if the set of detected flaws in the projection of the task *TSK* contains an *unachieved top-level command bug*. There are two ways in which an achievement task can fail to achieve a goal *g*. The rst

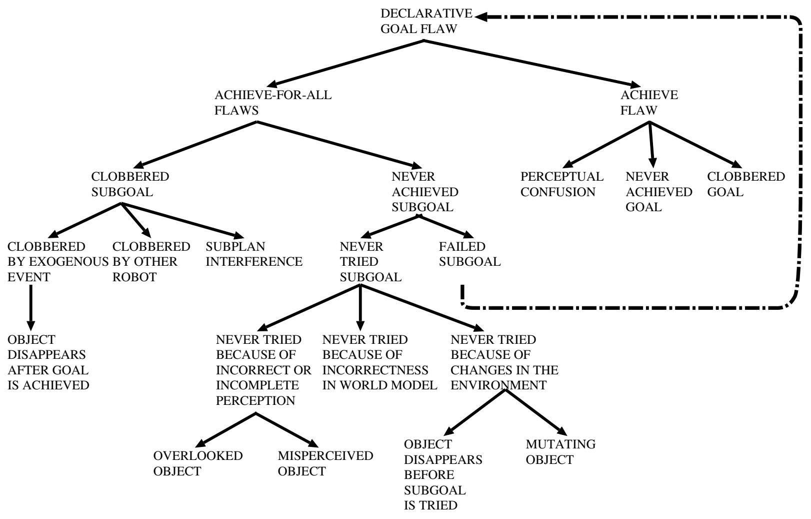


Fig. 3.10. Behavior flaw taxonomy for achievement tasks for the delivery application in our simulated world.

possibility is that the plan never achieved the goal G (*never achieved goal*), the second that G has been achieved but ceased to hold before the task end (*clobbered goal*). In order to determine whether a behavior flaw is of the type *never achieved goal* or the type *clobbered goal*, the planner has to check in the projected execution scenario whether G holds for some time during the projected plan execution. If the goal was never achieved, cases in which the goal was never tried (*never tried goal*) and others in which the achievement did not succeed (*achievement failure*) are distinguished. When an already achieved goal has been destroyed it, is often useful to distinguish whether the cause is a robot action (*task interference*) or an *external event*. In the case of a *task interference* the planner can set up protections or use mechanisms for goal reduction and the addition of ordering constraints on subtasks to avoid the behavior flaws. If the achievement of G failed (*achievement failure*), the planner could try to nd a subtask that did not work. The corresponding re- nement condition states that one of the subtasks did not satisfy its behavior constraint. In this case the planner does not specialize the flaw category but concludes that it is caused by a flaw in the subtask.

Organizing the plan revision methods of planning systems that anticipate and forestall behavior flaws of autonomous robots within such a taxonomy of behavior flaws has several advantages:

1. Even if the speci c class of a projected behavior flaw is not in the taxonomy the classi cation algorithm will nd a more general model that will describe the detected flaw in more general terms: any behavior flaw caused by a plan for an achievement goal is either of the type *never achieved goal* or *clobbered goal* and we have some very general plan revision methods stored for these failure models that might eliminate the projected behavior flaw.
2. The taxonomy contains very general models of behavior flaws and revisions to avoid them. For example, the planning operations performed by the classical general-purpose planning systems are are stored under the failure models *subplan interference* and *never tried (sub)goal*. Many other models of behavior flaws have the same kind of generality. The taxonomy covers behavior flaws caused by *imperfect sensors*, *imperfect control*, an *inaccurate and incomplete world model*, *subplan interference*, *exogenous events*, and *other robots* and applies to many autonomous robots.
3. If speci c robot applications require failure models and plan revision methods that are job, environment, and robot speci c the programmer can equip the robot with such very speci c planning capabilities the programmer by simply adding new models of behavior flaws as leaves to the taxonomy.

   For example, a programmer might want to equip a delivery robot that occasionally has to deliver china with some very speci c plan revision methods such as, if a robot is predicted to break some china during its delivery, then the robot should start earlier and be more careful while delivering the china, pad the china before you start, or take a less bumpy route. To do this, she might add a task-speci c failure model *"object broke during its delivery"* with the above mentioned plan revision methods to the taxonomy.

3.5 Summary and Discussion

Section 3.1 has described aspects of RPL, the language used to specify structured reactive controllers, that qualify it as a plan language. Robot plans are robot control programs that cannot only be executed but also reasoned about and manipulated. The XFRM framework provides a toolkit that enables planning processes to project what might happen when an RPL program gets executed, to infer what might be wrong with a robot controller given an execution scenario, and to perform complex and meaningful revisions of structured reactive controllers.

The planning processes for anticipating and forestalling behavior flaws work according to the criticize-revise control strategy (see section 3.2). The criticize-revise control strategy is a variant of the generate/test/debug control strategy [151]. Unlike the the generate/test/debug control strategy, which aims at producing correct plans, the criticize-revise control strategy tries to revise plans to increase their expected utility. To do so the strategy tries to detect behavior flaws in projected execution scenarios and diagnoses the causes of these flaws. The detected and diagnosed flaws point the planner to plan revisions that might increase the expected utility of the plan. As in the generate/test/debug control strategy, the plan candidates proposed by the revision step are projected to evaluate their utility.

The diagnosis of many types of behavior flaws and the application of many plan revisions requires the planning system not only to reason about the physical e ects of plan execution but also the process of plan interpretation and the relations between both. To infer that a possible cause for an execution failure is an incorrect object description, the planner has to reconstruct the value of the RPL variable that should contain the object description and check whether this description describes the object correctly. Or, to diagnose a behavior flaw caused by a imperfect sensor error, the planner has to compare the value returned by a sensing routine with the value that a perfect sensor would return. Therefore, a projected execution scenario in the XFRM framework contains a timeline that represents the physical e ects of the robot's actions as well as the task network that enables to reconstruct the computational state of plan execution at any timeinstant.

For the purpose of improving the average performance of structured reactive controllers by anticipating and forestalling behavior flaws caused by routine plans it su ces to perform a small number of Monte Carlo projections for each candidate plan a planning e ort generates. This is important because Monte Carlo projection algorithms are the only existing projection algorithms that can predict the probabilistic and temporally complex behavior caused by structured reactive controllers. I have also argued that a small number of Monte Carlo projections su ces because routine plans exhibit in a large variety of (standard) situations a stable behavior.

Using taxonomies of behavior flaws and plan revision methods for avoiding them enables the programmer of an embedded robot planning system to start with a general taxonomy of behavior flaws of autonomous robots and to add speci c classes of behavior flaws when necessary. In the previous section we have described a taxonomy of behavior flaws that classi es behavior flaws caused by *imperfect sensors*, *imperfect control*, an *inaccurate and incomplete world model*, *subplan interference*, *exogenous events*, and *other robots*.

4. Transparent Reactive Plans

A robot that forestalls flaws in its intended course of action while carrying out its jobs must perform difficult and complex computations at a speed that is fast compared to the time that it takes to execute plans. The computational processes must infer the purpose of subplans, find subplans with a particular purpose, automatically generate a plan that can achieve some goal, determine flaws in the behavior that is caused by subplans, and estimate how good the behavior caused by a subplan is with respect to the robot's utility model. The computational problems these processes try to solve are, in their general form, unsolvable, or at the very least computationally intractable.

To make the computational problems feasible, we will focus in this chapter on ways to specify structured reactive plans so that it will be easier to reason about and manipulate them. The improvements will be accomplished by

– declarative statements that facilitate the inferences listed above; and
– enforcing properties of structured reactive plans and plan revision rules that assure that XFRM will generate only plans for which the inferences listed above can be performed fast.

The chapter consists of three parts. The first part describes declarative statements that make subplans and data structures transparent. The second part describes how the planner can enforce that its plans satisfy properties that enable the planner to apply simple and fast methods for failure diagnosis, plan revision, and plan manipulation. This part also shows how structured reactive plans have to be implemented to satisfy these properties. The last part of the chapter illustrates the use of declarative statements and gives examples of plans that are contained in the plan library of the delivery robot.

4.1 Declarative Statements in Reactive Plans

Classical planning started with goals, declarative descriptions of *what* should be done, and produced plans, procedural descriptions of *how* it should be done. To do something similar for autonomous robots acting in changing and partly unknown environments, it is necessary to treat the achievement or maintenance of a state as specifying a routine behavior. At the same time we have to deploy tools to generate better behaviors. In this section we extend RPL with *declarative statements*, which will provide this functionality.

The extensions of RPL contain constructs like (ACHIEVE *P*) and (MAINTAIN *P*), (PERCEIVE *P*), or (PREVENT *P*). We use these constructs to make structured reactive plans transparent to automated planning algorithms. In other words, we represent the structure of plans and the purpose of subplans explicitly. To accomplish transparency we represent goals, perceptions, and beliefs as logical expressions [19]. Goals like (ACHIEVE *P*) or (MAINTAIN *P*) describe states of the world the robot has to achieve or maintain. Perception statements, like (PERCEIVE *D*), take a perceptual description *D* and return designators that describe objects satisfying *D* and thereby acquire new information. Belief statements (BELIEF *B*) make control decisions based on internal data structures transparent.

Let us illustrate the use of these extensions with an example. Below is a simple RPL plan for getting an object, specified by the designator DESIG, to the location ⟨*X-D, Y-D*⟩. The plan does not contain declarative statements.

```
(DEF-INTERP-PROC ACHIEVE-OB-AT-LOC (DESIG X-D Y-D)
  (COND ((= (DESIG->LOC DESIG) ⟨X-D,Y-D⟩)
         (NO-OP))
        ((AND (IS-HAND (DESIG->LOC DESIG))
              (= (ROBOT->LOC) ⟨X-D,Y-D⟩))
         (UNHAND (DESIG->LOC DESIG)))
        ((AND (IS-HAND (DESIG->LOC DESIG))
              (NOT (= (ROBOT->LOC) ⟨X-D,Y-D⟩)))
         (CARRY-OB-TO-LOC DESIG X-D Y-D))
        (T (SEQ (ACHIEVE-OB-IN-HAND DESIG)
                (CARRY-OB-TO-LOC DESIG X-D Y-D)))))
```

The plan ACHIEVE-OB-AT-LOC works as follows. If, according to DESIG, the object is already at its destination (= (DESIG->LOC DESIG) ⟨*X-D,Y-D*⟩), ACHIEVE-OB-AT-LOC will succeed without doing anything (NO-OP). If DESIG specifies that the object is in one of the robot's hands (IS-HAND (DESIG->LOC DESIG)) and the robot is at the object's destination (= (ROBOT->LOC) ⟨*X-D,Y-D*⟩), the robot will just open the hand holding DESIG. If the robot is not at ⟨*X-D,X-D*⟩ but holds the object in its hand, the robot will carry the object to its destination. Otherwise, (if DESIG is neither at its destination nor in one of the robot's hands,) the robot will first get the object and then carry it to ⟨*X-D, Y-D*⟩.

While for us, as programmers, this plan is fairly easy to understand (for one reason, the names of the subplans tell us what they do), it is extremely difficult for a planning system to automatically infer the teleological structure of this plan and the purpose of its subplans. Using declarative statements, we can restate the RPL plan ACHIEVE-OB-AT-LOC as follows:

```
(DEF-GOAL-PROC (ACHIEVE (LOC DESIG ⟨X,Y⟩)
    (COND ((BELIEF (LOC DESIG ⟨X,Y⟩))
             (NO-OP))
            ((BELIEF (AND (HAS ROBOT DESIG))
                            (LOC ROBOT ⟨X,Y⟩)))
             (ACHIEVE (ON-GROUND DESIG)))
            ((BELIEF (AND (HAS ROBOT DESIG))
                            (THNOT (LOC ROBOT ⟨X,Y⟩))))
             (WITH-POLICY (SCOPE NOW* (BEGIN-TASK LET-GO)
                                (MAINTAIN (HAS ROBOT DESIG)))
                 (SEQ (ACHIEVE (LOC ROBOT ⟨X,Y⟩))
                       (ACHIEVE (ON-GROUND DESIG)))))
            ((BELIEF (AND (THNOT (LOC DESIG ⟨X,Y⟩))
                            (THNOT (HAS ROBOT DESIG))))
             (WITH-POLICY (SCOPE OBJECT-GRASPED*
                                      (BEGIN-TASK LET-GO)
                                (MAINTAIN (HAS ROBOT DESIG)))
                 (SEQ (ACHIEVE (HAS ROBOT DESIG))
                       (ACHIEVE (LOC DESIG ⟨X,Y⟩)))))))).
```

The transparent version of the plan ACHIEVE-OB-AT-LOC has the name (ACHIEVE (LOC DESIG ⟨X,Y⟩)), which represents the intended purpose of the plan: make it true that the object designated by the data structure DESIG will be at the location $\langle X, Y \rangle$. (ACHIEVE (LOC DESIG ⟨X,Y⟩) could be implemented as follows: based on its "beliefs," the robot uses different plans to get DESIG to its destination. If the robot believes the object is at location $\langle X, Y \rangle$ it will do nothing. If the robot believes it is at location $\langle X, Y \rangle$ and has the object in its hand it will put the object on the ground. If the robot believes it has the object in its hand but is not at $\langle X, Y \rangle$ it will go to $\langle X, Y \rangle$ with the policy to keep the object in its hand, and will put the object on the ground at $\langle X, Y \rangle$.

In the transparent version of the plan, the subplans' names specify the type of the subplan (ACHIEVE and MAINTAIN) and the aspect of the world that is affected by the subplan (for instance, (LOC ?D ⟨X,Y⟩)). Functions that access data structures (for instance, designators) and sensor readings (for instance, the robot's current position provided by the that is updated by the robot's odometer) are made declarative using BELIEF-statements. For example, consider the RPL expression (= (DESIG->LOC DESIG) ⟨X,Y⟩) in the plan ACHIEVE-OB-AT-LOC, which evaluates to "true," if the location entry of the data structure DESIG specifies that the object designated by DESIG is at location $\langle X, Y \rangle$. In the corresponding conditional branch the robot acts as if it "believed" that the object designated by DESIG is at $\langle X, Y \rangle$ (see [142]). The meaning of the expression can therefore be represented explicitly as (BELIEF (LOC DESIG ⟨X,Y⟩)).

If the programmer uses the same predicates as arguments of declarative statements that the projector uses to describe the state of the world, she can specify the purpose of a plan in terms that can be easily "understood" by a planning system. That is, the planning system can infer the purpose of a declarative plan based on a syntactic analysis of its name.

So far we have used declarative statements only as a naming convention for subplans [121]. What differentiates declarative RPL statements like (ACHIEVE (LOC DESIG ⟨*0,10*⟩)) from other calls to subplans like (GET-OB-TO-LOC DESIG ⟨*0,10*⟩), however, is that declarative statements satisfy additional conditions: namely that they have routine plans, behavior specifications, and benefit models.

These pieces of information provided by declarative statements enable XFRM to perform several types of inferences on declarative statements and the RPL plans that contain them. For instance, because (ACHIEVE *S*) is a declarative goal, XFRM can make the following inferences: given a statement (ACHIEVE (LOC DESIG ⟨*X,Y*⟩)), XFRM can infer the answer to the question "was (ACHIEVE (LOC DESIG ⟨*X,Y*⟩)) successful?" by checking whether the state (LOC *OB* ⟨*X,Y*⟩) holds in the timeline for the object *OB* that is designated by DESIG upon completion of (ACHIEVE (LOC DESIG ⟨*X,Y*⟩)). We will see in the next chapter how XFRM can infer which object a designator designates. XFRM can also guess why (ACHIEVE (LOC DESIG ⟨*X,Y*⟩)) was not successful. By checking whether (LOC *OB* ⟨*X,Y*⟩) has held for some time during the execution of (ACHIEVE (LOC DESIG ⟨*X,Y*⟩)) XFRM can determine that (LOC *OB* ⟨*X,Y*⟩) did not hold in the end because it was clobbered after being achieved or because it has never been achieved. XFRM can also estimate how well the robot performed by comparing what it did with what it was supposed to do. Conversely, given the state (LOC *OB* ⟨*X,Y*⟩) XFRM can infer whether the robot has tried to achieve (LOC *OB* ⟨*X,Y*⟩) by checking whether it has executed a statement (ACHIEVE (LOC DESIG ⟨*X,Y*⟩)) such that DESIG designates *OB*. Declarative goals also facilitate many types of plan revisions: XFRM "knows" that the name of the plan for achieving *S* is (ACHIEVE *S*) and can simply add (ACHIEVE *S*) to the plan if needed.

XFRM uses declarative statements for failure diagnosis and plan revision in structured reactive plans: goals are used used for identifying failed tasks in execution scenarios and explaining the causes of failures, like imperfect control, task interference, or external events. Perception statements enable the planner to diagnose and forestall failures caused by imperfect and faulty sensor data. Belief statements are used to identify data structures that contain incorrect information about the world and therefore cause inappropriate control decisions and incorrect instantiations of procedure calls.

Many reasoning problems as, for instance, the problem of determining whether the robot has tried to achieve a certain state, are, for arbitrary structured reactive plans, unsolvable. But they essentially become a matter of pattern-directed retrieval if the parts of the plans needed to solve them are implemented using declarative statements.

4.1.1 RPL Construct Descriptions

The information that declarative RPL statements provide is bundled into RPL *construct descriptions*, which contain the syntactic form of declarative

Slot	**Content**
Routine Plan	RPL procedure definition of a routine plan for the goal.
Alternative Plans	RPL procedure definitions for alternative plans for the goal.
Benefit Models	A method for assigning a benefit to the behavior of the robot based on how well the behavior accomplished the goal.
Behavior Specifications	A method for determining flaws in the behavior of the robot with respect to accomplishing the goal.

Fig. 4.1. Structure of RPL construct descriptions.

statements, routine and alternative plans for declarative statements, benefit models, and behavior specifications.

Representation of Declarative Statements. Declarative goals and other declarative statements that describe jobs in cooperative and not completely structured environments have to be more expressive than declarative goals in completely known and static environments. In the latter case a declarative goal (ACHIEVE G) simply means performing a course of action that results in a world state that satisfies G and preserving G until the end of the plan. In cooperative and changing environments, however, we have to be more precise because in such a context (ACHIEVE G) means achieve the state G *before*, and maintain it *until*, it is needed.

To illustrate that these two aspects of achievement goals are important, consider two robots that are supposed to cooperate. One should unload a truck and the other put the unloaded objects where they belong. In classical planning, we would specify the job as "achieve that all objects currently in the truck will be beside of the truck." If, however, the second robot puts the objects in their final destinations while the first robot is unloading the truck, the goal state of the first robot, that all objects are beside the truck, will never be achieved because once the truck is unloaded some of the objects have already gone to their final destinations. In this case, the first robot has accomplished its job despite the fact that the goal state is never achieved. In another context, if the first robot is told that the second robot starts putting the objects at their destinations at three o'clock, it might not be enough to achieve that all objects are beside the truck, but rather the first robot has to make sure they stay there until three o'clock.

In order to enable the robots to handle their jobs in different contexts, we will extend the syntax of declarative statements with additional parameters that can express these different kinds of jobs. We will explain these extensions in the discussions of the different types of declarative statements.

Routine and Alternative Plans. It is no use having a plan step of the form (ACHIEVE G) unless the system actually knows how to make G true. This is especially true in a reactive system, which cannot tolerate nonexecutable plan fragments. Although goals are traditional, we run into trouble when we attempt to adopt the traditional techniques for generating plans to achieve them. These methods typically start from a description of a set of *operators* with propositional consequences (for example, add lists and delete lists). An operator is relevant to a goal if it adds propositions involving the predicates mentioned in the goal. A plan is typically found by proposing an operator to solve part of the goal, then embedding that operator in a plan for the rest of the goal [125]. Unfortunately, the choice of operators and embeddings is not obvious, so we quickly get an intractable search problem [45].

The best approach we see to reactive goal decomposition is to assume that for every goal G there is a routine plan for the job (ACHIEVE G) that can always be executed (and will, in benign circumstances, actually cause G to become true). If there is no time to think, this is the plan that gets executed. When there is more time, the planner can look for better plans.

Benefit Models. RPL construct descriptions also provide benefit models that measure how well the robot has achieved a top-level command. Given an execution scenario and a top-level command, the planner uses the benefit model of the top-level command and applies it to the execution scenario to determine the benefit obtained from accomplishing the top-level command. Benefit models of declarative statements can handle temporal aspects and partial achievements of top-level commands and balance the benefit trade-offs between top-level commands.

Behavior Specifications. Benefit models measure how well a plan accomplishes the top-level commands, but they cannot tell the planner how a plan can be improved. The latter is the job of behavior specifications. Behavior specifications are models of the intended behavior of the robot and the world. These models of intended behavior can be compared to the projected behavior. The differences between both behaviors are represented as *behavior specification violations* and describe flaws in the robot's behavior.

Figure 4.2 shows the role of behavior specifications in the detection of flaws of the robot's behavior. The planning process retrieves the behavior specifications of the given top-level commands and compares them with the projected execution scenario.

Let us explain the difference between benefit models and behavior specifications using an example. Suppose the robot is given a top-level command C, "get all balls from location $\langle 5,5 \rangle$ to $\langle 2,5 \rangle$ before 3pm," and an execution scenario ES. The benefit model of the top-level command returns a numerical value that describes the benefit that the robot has obtained from the way it has accomplished C. For instance, in execution scenario ES the robot might have earned a benefit of 138 for accomplishing C. Ideally, the robot could have received a benefit of 300. Using the behavior specification XFRM detects flaws such as BALL-3 is not at $\langle 2,5 \rangle$ at the end of the plan execution, or that the plan for accomplishing C was not completed before 3pm.

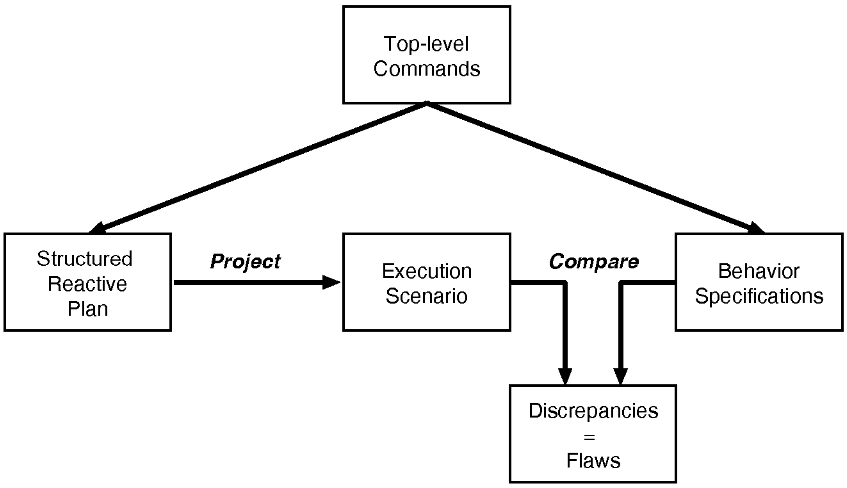


Fig. 4.2. The role of behavior specifications in the detection of flaws in the robot's projected behavior.

4.1.2 Achievement Goals

An important category of declarative statements are achievement goals, which allow us to express delivery jobs, painting jobs, and so on. In this section we describe achievement goals by means of their syntax, behavior specifications, and benefit models. Routine plans for declarative statements are described and discussed in section 4.3.3.

ACHIEVE. The basic achievement job has the form (ACHIEVE *G*) where *G* is a proposition or a conjunction of propositions describing constraints on a desired state of the world or the robot. Typical goals are (LOC ROBOT ⟨*2,5*⟩), (LOC *BOX-453* HAND-2), and (COLOR *BALL-56* RED). Some terms like ROBOT and HAND-2 are objects in the world (the robot controller can refer to the robot and the robot's hand correctly and uniquely) while other terms like *BOX-453* and *BALL-56* are descriptions of objects that can be incorrect and ambiguous.

In order to gain the expressiveness necessary for formulating the achievement jobs we have discussed earlier in this chapter, we extend the syntax of ACHIEVE with the optional keyword parameters :BY and :UNTIL. Thus, ACHIEVE-statements have the general form (ACHIEVE *G* :BY EV_1 :UNTIL EV_2) where EV_1 specifies a deadline and EV_2 the time span for which *G* has to be maintained. The arguments for :BY and :UNTIL are event specifications of different kinds: the passing of a deadline (:BY (TIME *14:00*)), a certain state of plan interpretation (:BY (TASK-END *T*)), or an event caused by another robot (:BY (*BEGIN* (DOES ROBOT3 PICK-UP DESIG)).

:UNTIL parameters can be used to represent *protections* [156] and causal links [119] that are well-known in classical planning. For instance, we can represent a plan for achieving a conjunctive goal in RPL as follows:

```
(REDUCE (ACHIEVE (AND (G1 G2)))
   (:TAG CG (PLAN
                 (ACHIEVE G1 :UNTIL (TASK-END CG))
                 (ACHIEVE G2 :UNTIL (TASK-END CG))))).
```

To achieve a conjunctive goal (AND G_1 G_2), the robot must first achieve one goal and then preserve it until the other goal is also achieved. We specify the plan for accomplishing (AND G_1 G_2) by reducing (ACHIEVE (AND G_1 G_2)) into a plan that consists of two concurrent subplans, one for accomplishing each subgoal. As you might recall, the statement (REDUCE *G P*) specifies that the plan step G is carried out by performing the plan P (see chapter 3.1.2). The task of interpreting the PLAN-statement is "tagged" with the name CG in order to enable the :UNTIL-arguments to refer to the completion of the task CG.

Similarly, we can implement protections or causal links between two plan steps S_1 and S_2 by stating that the robot has to achieve a goal G before it starts executing S_2 and make sure that G persists until S_2 starts. Or, we can represent goals in a cooperative setting by

```
(ACHIEVE (LOC DESIG ⟨0,8⟩)
     :UNTIL (BEGIN (DOES ?ROB PICK-UP DESIG))),
```

which means the robot has to get the object described by DESIG to location ⟨*0,8*⟩ and make sure that it stays there until a second robot picks it up.

We have added one more notation that has proven to be convenient for handling quantified goals. Often goals come up in a course of action as instances of quantified goals. For instance, the robot wants to deliver a set of objects one by one. This can be expressed as follows:

```
(LET ((OBS DESIGNATORS))
   (LOOP
    UNTIL (NULL OBS)
      (ACHIEVE ((λ (D) (LOC D ⟨X,Y⟩)) (FIRST OBS)))
      (!= OBS (REST OBS)))).
```

The behavior specifications of ACHIEVEMENT-statements specify how the world is to change as the ACHIEVEMENT-statement is executed. Consider the declarative goal (ACHIEVE S :BY EV_1 :UNTIL EV_2) and let T_1 be the time instant at which EV_1 occurs and T_2 be the one at which EV_2 occurs. Then the behavior specification specifies that goal G will have to hold from T_1 to T_2, i.e., (HOLDS G (TT T_1 T_2)). A behavior specification violation is the difference between the execution scenario and the "closest" execution scenario in which the ACHIEVE-job is accomplished: (THNOT (HOLDS G (TT T_1 T_2))).

The second tool is the benefit estimator which is used for judging how well the robot accomplished a job of the form (ACHIEVE S :BY T :UNTIL EV).

The benefit estimation depends on goal G, and takes penalties for missing deadlines and not maintaining the goals long enough into account.

The third piece of information associated with ACHIEVE-statements are routine plans. A routine plan is defined by a particular kind of procedure definition and stored in the plan library. Routine plans might ignore parameters :BY or :UNTIL; in this case the parameters are merely annotations for plan failure diagnosis. In order to achieve the goal (ACHIEVE (LOC DESIG ⟨*0,12*⟩)), the RPL interpreter retrieves the procedure definition for (ACHIEVE (LOC *?OB ?LOC*)) and executes the body of the procedure with *?OB* bound to DESIG and *?LOC* bound to ⟨*0,12*⟩.

ACHIEVE-FOR-ALL. Another important aspect of declarative goals is their quantification. We would like to give our robots jobs like *"get all the balls from ⟨0,8⟩ to ⟨0,10⟩*, or *"get three empty boxes."* One might be tempted to view a universally quantified goal as (ACHIEVE $\forall x.(D(x) \supset G(x))$). A problem with viewing quantified goals this way is that the robot could simply make *D(x)* false and thereby satisfy the implication without accomplishing the desired job. We can solve the problem by adding temporal components to $D(x)$ and $G(x)$.

In RPL, universally quantified achievement goals are specified as a separate RPL command ACHIEVE-FOR-ALL that has the syntactic form

(ACHIEVE-FOR-ALL D G :BY EV_1 :UNTIL EV_2 :DOMAIN-AT T).

The first argument of the ACHIEVE-FOR-ALL statement is a description D of the objects for which the goal G has to be achieved. The intention of the RPL construct ACHIEVE-FOR-ALL is that all objects which satisfy the property D have to satisfy the property G. :DOMAIN-AT can be used to constrain the time when the objects have to satisfy the given description (which can be either a time point or an interval). For instance, (:TAG J (ACHIEVE-FOR-ALL D G :DOMAIN-AT (DURING J))) specifies that G has to be achieved for each object that satisfies D for some time during the performance of J. Or, (ACHIEVE-FOR-ALL D G :DOMAIN-AT (BEGIN-TASK TOP-TASK*)) specifies that G has to be achieved for each object that satisfies D in the beginning of the execution of the top-level plan.

4.1.3 Perceptions

A robot senses its environment to obtain more information and perform its actions more robustly. The sensing operations are represented in RPL plans with the PERCEIVE-statement. The basic form of the PERCEIVE-statement is (PERCEIVE *STH*), where *STH* could be an object, an object description, an event, or a state. In the case of an object or an object description, the perception statement is intended to return a designator, a data structure that (partially) describes a perceived object and can be used to find the object again and manipulate it. The perception statement returns the value FALSE, if no object satisfying the given description can be perceived.

Like ACHIEVE-statements, PERCEIVE-statements come with several keyword parameters: :EXAMINE, :TRACK, and :UNIQUE. The statement

```
(PERCEIVE DESIG :EXAMINE PROPS)
```

is used to obtain information about an object. The subplan asks the robot to take an image of the object designated by *DESIG* and examine its properties *PROPS*. The designator returned by the PERCEIVE-statement is an updated version of *DESIG* that describes the properties *PROPS* of the object. The second use of the PERCEIVE-statement is tracking objects in order to manipulate them. The statement (PERCEIVE *DESIG* :TRACK YES) finds the object described by *DESIG* and starts tracking it. If the keyword parameter :UNIQUE is passed as an argument to the perception subplan, the perception statement should return a failure if the object description *DESIG* is ambiguous, i.e., if more than one object satisfies the description *DESIG*.

Perception statements also come in a universally quantified version: (PERCEIVE-ALL *D*) returns for each object that satisfies the description *D* one designator describing the object. Each designator describes exactly one object and the features stored in the designator correctly describe, as far as the robot knows, the features of the object. Scene recognition can be realized by the perception statement

```
(PERCEIVE-ALL (λ (O) (LOC O ⟨X,Y⟩))
                 :EXAMINE (CATEGORY COLOR SIZE)).
```

The behavior specifications of perception statements specify criteria for the correctness and completeness of the observations. The benefit models of perception statements try to measure the value of the information obtained by a perception.

4.1.4 Beliefs

The main purpose of data structures in RPL plans is to store information about aspects of the world that have been sensed and to use this information in order to control which subplans to execute and how subplans are to be instantiated. The following piece of RPL code illustrates this point.

```
(LET ((⟨X,Y⟩ (DESIG->LOC OB)))
   (IF (AND (DEFINED? X) (DEFINED? Y))
       (GO ⟨X,Y⟩)
       (FAIL :UNKNOWN-OBJ-LOC OB)))
```

The local variables X and Y are set to the X- and Y-coordinates that are specified by the designator OB. The body of this plan fragment specifies that if the values of the variables X and Y are position coordinates, go to the location with the coordinates X and Y; otherwise fail. Therefore, the intended meaning of this piece of code is "go to the location of object OB; if you do not know where it is, fail."

We have introduced the statement (BELIEF *P*) into RPL in order to make these beliefs transparent. The statement (BELIEF *P*) returns true and binds the variables that occur in *P* if the robot "believes" *P*. If the BELIEF-statement occurs as a condition in an IF-statement, the logical variables that occur in *P* are local variables of the THEN-subplan and are initialized with the binding of the variables for which *P* is "believed." Using the BELIEF-statement, we can transparently represent that the variables X and Y are the robot's "beliefs" about the location of the object described by OB.

```
(IF (BELIEF (LOC OB ⟨?X,?Y⟩)))
    (ACHIEVE (LOC ROBOT ⟨X,Y⟩))
    (FAIL :UNKNOWN-OBJ-LOC OB))
```

We have not yet explained how the belief statements are grounded in the data structures used by the structured reactive plan. The correspondence between predicates and data structures is specified by the plan writer using PROLOG-like rules. The following Horn clause rule specifies that if ?X is the content of the slot X-COORD and ?Y the content of the slot Y-COORD of the data structure ?OB then the robot believes the object described by the designator ?OB is at location ⟨?X,?Y⟩.

```
(DEF-FACT-GROUP BEL-LOCATION
   (PROLOG (BELIEF (LOC ?OB ⟨?X,?Y⟩))
           (LISP-PRED IS-DESIG ?OB)
           (EVAL (DESIG-GET ?OB 'X-COORD) ?X)
           (EVAL (DESIG-GET ?OB 'Y-COORD) ?Y)
           (THNOT (== ?X FALSE)) (THNOT (== ?Y FALSE))))
```

First-order logical statements are a very convenient representation of beliefs for the purpose of planning. The planner can simply take a belief *B* and generate the deductive query (HOLDS *B* (TASK-END *TSK*)) in order to check whether *B* holds in a given execution scenario. In addition, the update of the belief *B* can be realized by adding the statement (PERCEIVE *B*) to the plan. The basic mechanisms necessary to reconstruct beliefs at any time instant during a projection are discussed in [127].

4.1.5 Other Declarative Statements

We have not yet mentioned maintenance and stabilization goals, two other declarative statements used for controlling the delivery robot. As subroutines, maintenance goals are often used in the context of policies. For instance,

```
(WITH-POLICY (MAINTAIN (HAS ROBOT OB))
   (ACHIEVE (LOC ROBOT ⟨X,Y⟩)))
```

specifies that the robot must hold on to the object while going to location $\langle X, Y\rangle$. An RPL plan for achieving this behavior could ask the robot to stop going whenever the hand force sensor drops to zero, pick up the lost object, and continue on to $\langle X, Y \rangle$.

Another important class of declarative goals are stabilization goals. Stabilization goals [87] make routine activities more robust and efficient by actively stabilizing aspects of the environment. An example of a stabilization goal is putting glasses back in the cupboard after using them. A household robot can spend effort on keeping glasses in the cupboards even if it is not asked to by explicit commands. If glasses are kept in the cupboard, other routines that make use of glasses do not have to waste time to find them.

Another declarative construct is (AT-LOCATION *LOC PLAN*), a macro specifying that the plan *PLAN* has to be executed at a particular location *LOC*. The XFRM scheduler uses this declarative statement to find out where the robot has to go to perform a certain task and to place ordering constraints on structured reactive plans [127].

The plan writer has to decide which plans in the plan library are essential for plan revision and should thus be implemented transparently using declarative statements. She can add new declarative statements to the plan language by specifying RPL construct descriptions for them. For instance, a plan writer can provide a new declarative statement (VERIFY *P*), which has the following meaning: if (PERCEIVE *P*) returns FALSE (this means *P* does not hold provided that the sensor program works correctly), then fail; otherwise succeed. The advantage of having a VERIFY-statement is that the planner can automatically add and delete verification steps in a plan. It can add a verification step if a plan subplan is not robust in achieving a property *P* and does not recognize its failures. It can delete the verification step if the plan for (PERCEIVE *P*) turns out to be unreliable or if the plan as a whole is too slow and the planner is willing to take the risk that failures of the type "state *P* does not hold" are no longer detected at execution time.

4.1.6 Using Declarative Statements

In order to use declarative statements as abstractions of subplans, structured reactive plans must contain subplans with purposes that can be described in terms of states of the environment and the robot. Fortunately, this requirement is the same as one of the basic design principles for behavior-based control systems (see Section 2.2.8 and [118]): behavior-based control systems are composed of control processes that try to achieve and maintain certain states of the robot and its environment. Another kind of subplan in structured reactive plans senses aspects of the environment in order to acquire information about the state of the world. The information that these subplans perceive can also be described in terms of logical statements. Finally, structured reactive plans keep data structures in global variables and fluents to store information about the state of the world. Consequently, the plan writer can name task-achieving behaviors and sensing routines with declarative goals and perception statements and abstract away from data structures using belief statements.

The feasibility of implementing structured reactive plans using declarative statements raises another issue: "how much of the structured reactive plan *can* and *should* be implemented in form of declarative statements?" This issue is best discussed using an example such as the plan (ACHIEVE (LOC ROBOT ⟨*X*,*Y*⟩)) that we have discussed on page 77. The representation of the plan using declarative statements does not have to stop at the level of abstraction shown on page 77. For example, the programmer can implement the plan (ACHIEVE (LOC ROBOT ⟨*X*,*Y*⟩)) that brings the robot to location $\langle X, Y \rangle$ using declarative statements: the robot moves towards its destination until it believes to be there; while moving, it stays away from obstacles.

```
(REDUCE (ACHIEVE (LOC ROBOT ⟨X,Y⟩))
    (WITH-POLICY (MAINTAIN (NOT-BUMPED-INTO-OBSTACLE))
        (FILTER (NOT (BELIEF (LOC ROBOT ⟨0,10⟩)))
           (MAINTAIN (MOVING-TOWARDS ⟨0,10⟩)))))
```

The programmer could even go a step further and provide a transparent plan for (MAINTAIN (NOT-BUMPED-INTO-OBSTACLE)): whenever the robot believes the location ahead is blocked, it will reach a state in which the robot is still next to the obstacle but unblocked towards its destination. This could look as follows:

```
(REDUCE (MAINTAIN (NOT-BUMPED-INTO-OBSTACLE))
    (WHENEVER (BELIEF (OBSTACLE-AHEAD))
         (WITH-POLICY (MAINTAIN (NEXT-TO-OBSTACLE))
             (ACHIEVE (UNBLOCKED-TOWARDS ⟨X,Y⟩)))))
```

The decision whether to implement the process of circumnavigating obstacles ((MAINTAIN (NOT-BUMPED-INTO-OBSTACLE))) using declarative statements is a pragmatic one. If the planner must revise subplans for circumnavigating obstacles, the decision will be positive. In the transparent version of the subplan for circumnavigating obstacles the planner can detect improper functioning of the routine for obstacle detection. It can also detect that getting the robot to a location that is unblocked towards the destination might not be sufficient to bring the robot to its destination if the obstacle has concavities. If some obstacles are concave a robust navigation algorithm has to get the robot to a location next to the obstacle that is closest to the destination [114], which, depending on the available sensors, might require the robot to go around the whole obstacle. On the other hand, if the robot does not have to revise its routine for circumnavigating obstacles, it is not necessary to implement the routine transparently.

4.2 Routine Plans

Structured reactive controllers need reliable and fast algorithms for the construction of routine plans, the diagnosis of behavior flaws, and editing subplans during their execution. Since these computational problems are for arbitrary RPL plans unsolvable, we will not use algorithms that work for arbitrary structured reactive plans. Instead we will use algorithms that make assumptions to simplify the computational problems. An attractive alternative to making assumptions about worlds and robots, as done by other planning algorithms, is making assumptions about plans. This is attractive because the planner constructs and revises the plans and can thereby enforce that the assumptions hold.

As a transformational planner, XFRM generates only a small subset of valid RPL plans: this set consists of the routine plans and their revisions (see figure 4.3). A routine plan for the commands $CMD_1, \ldots, CMD_i, \ldots, CMD_n$ has the form

```
(WITH-POLICY POL_1
   ...
     (WITH-POLICY POL_m
        (PARTIAL-ORDER
         ((:TAG MAIN
               (TOP-LEVEL
                 (:TAG CMD-NAME_1 ROUTINE-PLAN_1)
                 ...
                 (:TAG CMD-NAME_n ROUTINE-PLAN_n))))))),
```

if for each CMD_i there exists an entry in the plan library of the form (TO C_i R_i) and $CMD_i = C_i\theta_i$ where θ_i is a binding, that is, a mapping from variables in CMD_i to values, and $ROUTINE\text{-}PLAN_i = R_i\theta_i$. The *revised routine plans* are the reflexive, transitive closure of the routine plans with respect to the application of plan revision rules.

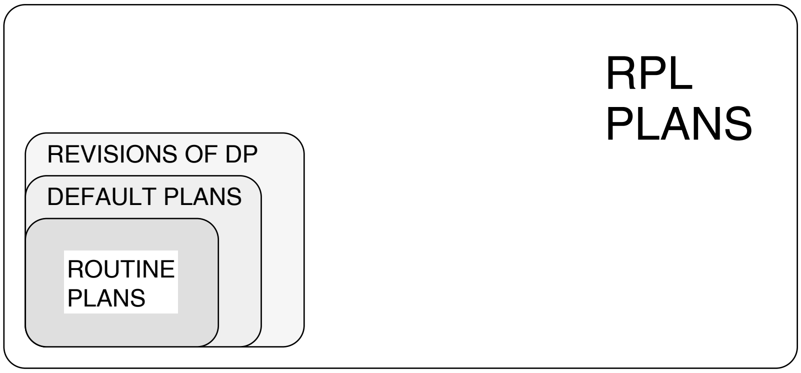


Fig. 4.3. The set of syntactically possible RPL plans and the set of plans that can be generated by a transformational planner.

The operational definition of the plan language is preferable over a syntactic definition because it suggests a way to enforce that the plans in the language have a certain property Q. Suppose we had, for a given property of plans, say Q, a method that maps any RPL plan and any plan revision rule into a semantically equivalent one that has or preserves the property Q.[1] We could then use such a method to rewrite a transformational planning system with a plan library $\mathcal{P}$ and a set of plan revision rules $\mathcal{R}$ into one with a plan library $\mathcal{P}'$ and revision rules $\mathcal{R}'$, which can generate the same behaviors of the robot and forestall the same kinds of behavior flaws but reasons only about plans that satisfy the property Q. If Q simplifies the computations listed above, we will can obtain simpler computational problems without limiting the scope of robot behaviors we can specify.

The Properties of Routine Plans. We propose that XFRM should only reason about structured reactive plans that are *transparent*, *restartable*, and *coherent*. Transparency enables the planner to understand important aspects of the plan by syntactic matching and pattern-directed retrieval. Restartability enables the robot to integrate a plan revision into a subplan by simply repeating the execution of the subplan. Coherence simplifies the construction of routine plans because coherent plans can run concurrently without additional synchronizations and still achieve a coherent problem-solving behavior. Below we will describe these properties more carefully and show how structured reactive plans can satisfy these properties.

Transparency. Inferring the purpose and structure of an arbitrary structured reactive plan is impossible. For this reason, we propose that plans provide a model of their purpose and structure that is sufficient for XFRM to diagnose execution failures. We will call structured reactive plans that provide such a model *transparent plans* and the corresponding property *transparency*. Transparent plans support a planning algorithm in inferring whether the robot has tried to achieve, maintain, or perceive a given state. They specify the purpose of subplans in terms of states of the environment (using declarative RPL statements) and enable planning modules to infer important aspects of the plan by syntactic matching and pattern-directed retrieval.

A plan is transparent with respect to achieving a state G if the plan satisfies the following two conditions: (1) For each call of a control routine C that has an intended effect G, there exists a plan P of the form (REDUCE (ACHIEVE G) $BODY$) such that $BODY$ contains C. (2) For each plan of the form (REDUCE (ACHIEVE G) $BODY$), $BODY$ is locally relevant for achieving G. A plan is locally relevant for achieving a goal G if a situation exists in which $BODY$ has a high probability of achieving G. Figure 4.4 shows a code tree of a transparent plan, highlighting the declarative subplans.

Let us illustrate transparency with an example. Suppose we have a plan which contains a call to a control routine $C =$ (GO $\langle X,Y \rangle$). If this plan is

[1] A plan P_1 is equivalent to a plan P_2 if executing P_1 causes the robot to exhibit (non-deterministically) the same behavior as it would by executing P_2.

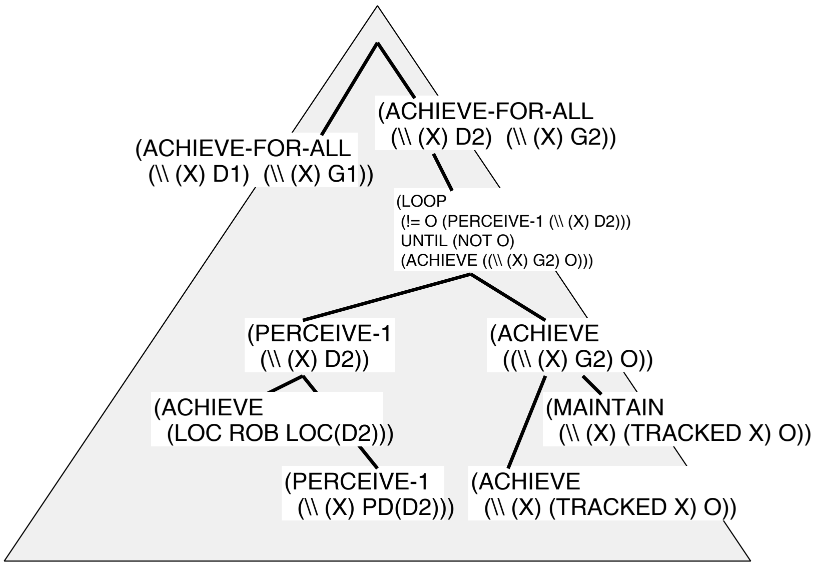

(ACHIEVE-FOR-ALL (\\ (X) D1) (\\ (X) G1))
(ACHIEVE-FOR-ALL (\\ (X) D2) (\\ (X) G2))
(LOOP (!= O (PERCEIVE-1 (\\ (X) D2))) UNTIL (NOT O) (ACHIEVE ((\\ (X) G2) O)))
(PERCEIVE-1 (\\ (X) D2))
(ACHIEVE ((\\ (X) G2) O))
(ACHIEVE (LOC ROB LOC(D2)))
(MAINTAIN (\\ (X) (TRACKED X) O))
(PERCEIVE-1 (\\ (X) PD(D2)))
(ACHIEVE (\\ (X) (TRACKED X) O))

Fig. 4.4. Code tree of a transparent RPL plan for delivery jobs.

transparent with respect to where the robot goes, it must contain a subplan of the form (REDUCE (ACHIEVE (LOC ROBOT $\langle X,Y \rangle$)) ϕ) such that ϕ contains C. Also, any plan of the form (REDUCE (ACHIEVE (LOC ROBOT $\langle X,Y \rangle$)) ϕ) has to bring the robot to location $\langle X,Y \rangle$ for some initial condition. Under these two assumptions, an RPL plan will try to get the robot to a certain location $\langle X,Y \rangle$ if and only if the task network contains a task with a plan of the form (REDUCE (ACHIEVE (LOC ROBOT $\langle X,Y \rangle$)) P). In this case, P is the plan for achieving (LOC ROBOT $\langle X,Y \rangle$)).

To be sure that we can implement transparent structured reactive plans, we must accomplish two things. First, we have to show that large parts of the structured reactive controllers can be organized and structured into subplans that achieve, maintain, and perceive states in the world, and make decisions based on computational states that are related to the states in the world. Second, we must confirm that by restating these subplans using declarative statements we are not changing the controllers in ways that affect the physical behavior of the robot. We have shown the first criterion in section 4.1.

In order to explain whether and under which conditions the second condition holds, let us visualize the semantics of an RPL plan as a tree of computational states that are produced by its interpretation. The multiple branches in the tree correspond to non-deterministic choices in the plan interpretation process. The interpretation of an RPL plan is a path in this tree from the root to a leaf. A computational state is characterized by the status of the tasks, the active behavior modules, and the RPL program environment of each task.

To investigate how the replacement of P with (REDUCE G P) will affect the physical behavior of the robot we have to look at the semantics of the REDUCE-statement and how it is interpreted by the RPL interpreter. Davis [52] gives an axiomatization of a robot control language similar to RPL. In such a semantics the status of the task with plan (REDUCE G P) is always the same as the status of the subtask for executing P. The active behavior modules of (REDUCE G P) are the active behavior modules of P. Substituting P with (REDUCE G P) changes nothing in the tree of computational states. It only introduces an additional task (REDUCE G P) that always has the same status as P.

The interpretation of the REDUCE-statement introduces a small overhead but we can neglect this overhead if we assume the interpreter to be infinitely fast [52]. This assumption is reasonable because the time for setting or testing the value of a program variable is negligible compared to the time needed for moving the robot's arm or going to another location.

Restartability. Restartability is another important property of structured reactive plans. Restartability means that the robot controller can repeatedly start executing a plan, interrupt the execution, and start it anew, and the resulting behavior of the robot is very similar to the behavior of executing the plan only once. Restartability facilitates the smooth integration of plan revisions into ongoing activities: a partially executed restartable plan can be revised by terminating the plan and starting the new plan that contains the revisions.

A plan P is *restartable* if for each subplan s the following conditions hold: (1) s is restartable or noninterruptable and (2) if s is not side-effect free then s has a guard *guard(s)* such that *guard(s)* is true if s has been executed.

Let P be a plan of the form $P_1 \circ P_2$ where P_1 is the part of the plan that is already executed and P_2 has not been executed yet. If P is restartable and the environment is static then the robot behavior caused by (SEQ P_1 P) is nondeterministically the same that is caused by P. If plans are restartable, a simple method for plan swapping suffices because the plan automatically skips over the parts it has already executed. Figure 4.5 shows RPL control structures and RPL plan idioms that make the control structures restartable.

In practice, however, restartable plans work differently. Consider, for example, a plan for the delivery of an object. To determine which parts of the plan can be skipped, it is sufficient to know the object's location and some aspects of the robot's state (like its location and what it is holding in its hands). Thus, if the structured reactive plans update the designators of the objects they perceive and manipulate, the robot controller can determine the state of execution based on the designator for the object to be delivered and the global variables and fluents of the structured reactive controller.

Restartable plans also simplify the projection of partially executed plans. Projecting what has already been executed is inefficient and, more importantly, a great source of errors: the projector might predict changes in the

Plan	Restartable Version
(LET ((OB '#F)) (LOOP (!= OB (PERCEIVE-1 (λ (X) D))) UNTIL (NOT OB) (ACHIEVE ((λ (X) G) OB))	(LOOP (IF (NOT *OB*) (!= *OB* (PERCEIVE-1 (λ (X) D)))) UNTIL (NOT *OB*) (ACHIEVE ((λ (X) G) *OB*)) (!= *OB* 'FALSE))
(SEQ A_1 ... A_n)	(SEQ (IF (NOT A$_1$-COMPLETED) (EVAP-PROTECT (:TAG A-ONE A_1) (IF (EQ (TASK-STATUS A-ONE) 'DONE) (!= A$_1$-COMPLETED '#T)))) ... (IF (NOT A$_n$-COMPLETED) (EVAP-PROTECT (:TAG A-N A_n) (IF (EQ (TASK-STATUS A-N) 'DONE) (!= A$_n$-COMPLETED '#T))))
(PAR A_1 ... A_n)	(PAR (IF (NOT A$_1$-COMPLETED) (EVAP-PROTECT (:TAG A-ONE A_1) (IF (EQ (TASK-STATUS A-ONE) 'DONE) (!= A$_1$-COMPLETED '#T)))) ... (IF (NOT A$_n$-COMPLETED) (EVAP-PROTECT (:TAG A-N A_n) (IF (EQ (TASK-STATUS A-N) 'DONE) (!= A$_n$-COMPLETED '#T))))

Fig. 4.5. RPL control structures and RPL plan idioms that make the control structures restartable.

world that are inconsistent with feedback that has been sensed during the execution of the corresponding subplans. If the planner projects a partially executed plan it is only interested in the part of the plan that has not been yet. If the planner projects a restartable plan, it will automatically skip the parts that already have been executed. Suppose a plan P has the form $P_1 \circ P_2$ where P_1 is the part of the plan that has already been executed and P_2 is the part that has not. If P is restartable (and the environment is static) then the execution scenario generated by projecting P_2 in the current state of plan interpretation is nondeterministically the same as generated by projecting P.

Coherence of Routine Plans. Coherence of routine plans can be achieved by associating valves with each sensor and effector and by requiring that each RPL process that needs access to a sensor and effector has to request the valve corresponding to the sensor or effector, block the valve until control over the sensor or effector is no longer required, and release the valve afterwards. Adding the additional synchronizations to a plan is a means of specializing the program.

In chapter 7 we will discuss another property of structured reactive plans, which specifies that interactions between subplans are made explicit. This dependency information facilitates the construction of local planning problems.

4.3 The Plan Library for the Delivery Application

This section gives a brief overview of the structure and plans contained in the plan library for the delivery robot in the DeliveryWorld.

4.3.1 Behavior Modules

The behavior modules (see chapter 2.2.2) of the delivery robot in the DeliveryWorld are variants of the "world actions" that have been developed and implemented by McDermott [127]: (ROBOT-START-MOVING *DIR*) is a behavior module that asks the robot to start moving to the adjacent location into direction *DIR*. If the location does not exist, nothing will happen. Upon completion the global fluent ROBOT-MOVED is pulsed. The behavior module (HAND-MOVE *H I*) causes the robot to move hand *H* to position *I* at the current location. When the move is completed the fluent HAND-MOVED is pulsed. When executing (GRASP *H*) the robot closes the gripper of hand *H*. The fluent HAND-MOVED is pulsed when the gripper is closed. If the grasp is successful the force measured by HAND-FORCE is high. (UNGRASP *H*) causes the robot to opens the gripper of hand *H*. The fluent HAND-MOVED is pulsed when the gripper is open. If an object is in the hand it will fall to the ground.

In addition, the delivery robot is equipped with several behavior modules for visual sensing. (LOOK-FOR-PROPS *PL*) takes an image with the camera and scans the image for objects satisfying *PL*. When the visual search is completed the fluent VISUAL-INPUT is pulsed. The global variable OBS-POS contains the positions of the objects satisfying *PL*. (POS-PROPS *I PROPS*) examines the properties *PROPS* of the object at position *I*. When the examination is completed, VISUAL-INPUT is pulsed. The variable OB-PROPS contains the values of the sensed properties. (START-TRACKING *I*) tracks the object at position *I*.

4.3.2 Low-level Plans

Low-level plans cause behaviors that in certain ranges of situations can robustly achieve their purposes. For example, the low-level plan PICK-UP works with high probability, if the object to be picked up is free to grasp, tracked by the robot's camera, and the robot's hand is empty. Other low-level plans examine objects, put down objects the robot is holding, and look for objects in sensor range that satisfy a given description (see figure 4.6). The planning modules consider low-level plans to be black boxes and use projection and effect rules (see chapter 3) to predict the behavior of the robot and the changes in the environment they cause. Again, the low-level plans used in this dissertation are variants of the ones originally described in [127].

Figure 4.7 shows the low-level plan LOOK-FOR that accomplishes the following behavior: given a perceptual description, return a designator for each

Plan	**Behavior**
(MOVE *DIR*)	Move to the next location in direction *DIR* (east, west, south, north).
(LOOK-FOR *PL*)	Look for an object that satisfies the list of properties *PL*. Return a designator for each matching object.
(EXAMINE *D PROPS*)	Examine the properties *PROPS* of the object specified by designator *D*. Returns the the sensed values of *PROPS*.
(PICKUP *D H*)	Pickup the object designated by *D* with hand *H*.
(UNHAND *H*)	Open the hand *H*, i.e., let the object that is held go.
(TRACK *D*)	Track the object designated by *D*.

Fig. 4.6. Low-level plans used by the delivery robot in the DeliveryWorld.

```
(DEF-INTERP-PROC LOOK-FOR (PL CAM)
  (SET-VALUE (TRACKING-FLUENT CAM) '())
  (:TAG LOOK-TASK
        (LET ((SEEN-OBS '())
              (SENSOR-PROG (AREF (DYNAMIC-VAL 'SENSOR-PROGS* LOOK-TASK) CAM)))
          (!= SEEN-OBS
              (LET ((KEPT-VISION (STATE '(KEPT-VISION)))
                    (SEE '#F)
                    (SEEN-OBS '())
                    (VISUAL-INPUT (AREF VISUAL-INPUTS* CAM)))
                (PROCESS LOOKER
                    (VALVE-REQUEST LOOKER WHEELS '#F)
                    (N-TIMES 3
                       (TRY-IN-ORDER
                           (WITH-POLICY (SEQ (WAIT-FOR KEPT-VISION)
                                             (WHENEVER (NOT KEPT-VISION)
                                                  (DBG-FORMAT ...)
                                                  (FAIL :VISION-SYSTEM-PRE-EMPTED)))
                               (VALVE-REQUEST LOOKER VISION KEPT-VISION)
                               (SETF (FLUENT-VALUE (AREF VISUAL-INPUTS* CAM)) '#F)
                               (LOOK-FOR-PROPS
                                 (MODEL->WORLD-PLIST PL) CAM SENSOR-PROG)
                               (!= SEE (WAIT-WITH-TIMEOUT
                                             (AREF VISUAL-INPUTS* CAM)
                                             (WORLD-SPEED-ADJUST (* 10 LOOK-TIME*))
                                             :KEEP-CONTROL))
                               (IF (NOT SEE) (FAIL :CLASS timeout))
                               (!= SEEN-OBS (SEEN-OB-DESIGS CAM PL))
                               (VALVE-RELEASE LOOKER VISION) )
                           (NO-OP))
                       UNTIL (AND SEE KEPT-VISION))
                    (VALVE-RELEASE LOOKER WHEELS))
                (IF (NOT KEPT-VISION)
                    (FAIL :VISION-SYSTEM-PRE-EMPTED)))
                SEEN-OBS)))
  (UPDATE-ENV-MODEL SEEN-OBS PL)
  (SET-VALUE (LOCAL-DESIGS-FLUENT CAM) SEEN-OBS)
  SEEN-OBS)))
```

Fig. 4.7. Low-level plan for looking for objects that satisfy a given perceptual description.

```
(DEF-INTERP-PROC TRACK (DESIG CAM)
   (IF (IS-LOCAL-DESIG DESIG CAM)
      (SEQ (START-TRACKING DESIG CAM)
           (SET-VALUE (TRACKING-FLUENT CAM) DESIG))
      (FAIL :CLASS nonlocal-designator :DESIG DESIG)))
```

Fig. 4.8. Low-level plan for tracking objects in the current camera image.

object in sensor range matching the given description. The kernel of the LOOK-FOR is a sequence consisting of the activation of the behavior module LOOK-FOR-PROPS and a second step that waits until the behavior module signals the completion of the visual search. The kernel is highlighted with the bold font.

The remaining part of the low-level plan makes the behavior more robust. Failures are signalled when the routine does not terminate within the given time frame, or when another process takes over the camera in the midst of the routine. The N-TIMES loop activates the behavior module LOOK-FOR-PROPS repeatedly until the behavior is completed and no interruptions have occurred, but at most three times. In order to avoid other routine activities using the camera or causing the robot to move to another location, the LOOK-FOR plan requests and blocks the valves CAMERA and WHEELS. The rest of LOOK-FOR deal with updating global fluents, updating the variable KNOWN-THINGS (UPDATE-ENV-MODEL SEEN-OBS PL), and computing the return value of the plan.

Another low-level plan tracks objects, that is, it keeps objects in the center of the camera view over extended periods of time (see figure 4.8). Physical operations performed on objects, such as picking up objects, putting them down, and modifying them, require the objects to be tracked by a camera. For the purpose of plan interpretation and planning, a track is a correspondence between a data structure used by the plan and an object in the world, that is established by an operation of the robot such as tracking or holding the object. We distinguish between visual and physical tracks. Visual tracking succeeds if the designator to be tracked designates an object in the current image. Visual tracks are destroyed by the robot moving away from the object such that it is out of sensor range, by taking new images with the camera, and tracking another object. There are also random exogenous events that cause the robot to lose visual tracks.

4.3.3 High-level Plans

Figure 4.9 shows a plan for bringing an object OB to a specified location $\langle X, Y \rangle$. The plan only distinguishes two forms of calls. In the first form the robot is the object asked to get somewhere, in the second one it is an object specified by a designator. Thus, the keyword parameters :BEFORE and :UNTIL do not affect the choice of the routine plan.

```
(DEF-GOAL-PROC (ACHIEVE (LOC OB ⟨X,Y⟩))
    (IF (MATCH GOAL '(LOC ?DES ⟨?X,?Y⟩))
        (SEQ (IF (EQ DES 'ROBOT)
                  (GO X Y)
                (PROCESS DELIVERY
                    (VALVE-REQUEST DELIVERY WHEELS '#F)
                    (SEQ (COORDS-HERE)
                          (IF (BELIEF '(LOC ,DES ⟨,X,,Y⟩))
                              (NO-OP)
                           (LET ((NEW-DES '#F))
                               (IF (BELIEF '(HAS ROBOT ,DES ?H))
                                   (NO-OP)
                                 (PROCESS GET-OBJECT
                                     (VALVE-REQUEST GET-OBJECT WHEELS '#F)
                                     (IF (BELIEF '(EMPTY-HAND ?H))
                                         (NO-OP)
                                      (LET ((H (RANDOM-LIST-ELEM (GLOB-EVAL HANDS*))))
                                          (ACHIEVE '(EMPTY-HAND ,H))))
                                          (IF (BELIEF '(EMPTY-HAND ?H))
                                              (!= DES (ACHIEVE '(HAS ROBOT ,DES ,H)))
                                            (FAIL :CLASS no-empty-hand-for :DESIG DES))
                                          (VALVE-RELEASE GET-OBJECT WHEELS)))
                               (IF (BELIEF '(HAS ROBOT ,DES ?H))
                                   (PROCESS CARRY
                                       (VALVE-REQUEST CARRY WHEELS '#F)
                                       (IF (BELIEF '(LOC ROBOT ⟨,X,,Y⟩))
                                           (NO-OP)
                                         (WITH-POLICY (MAINTAIN '(HAS ROBOT ,DES ,H))
                                             (ACHIEVE (LOC ROBOT ⟨X,Y⟩))))
                                       (UNHAND H)
                                       (VALVE-RELEASE CARRY WHEELS))))))
                    (VALVE-RELEASE DELIVERY WHEELS))))
     (ERROR-BREAK ACHIEVE-LOC :FATAL "ill-formed goal: a" GOAL)))
```

Fig. 4.9. High-level plan for getting an object to a specified destination.

The plan for the delivery of an object specified by a designator distinguishes between the following situations. First, the robot believes the object to be at the destination and therefore does nothing. Second, the robot gets the object in a first step, goes to a destination, while at the same time keeping the object in its hand, and lets go of the object. In the first step, the robot differentiates between three states: holding the objects in a hand, not holding the object but having a free hand, and having neither the object nor a free hand.

Figure 4.10 shows a routine transformation rule for goals of the form (ACHIEVE-FOR-ALL D G). A routine transformation is always applicable and produces a routine plan for the declarative statement it is applied to. A routine transformation rule has the form

$$\begin{array}{c} PL == PAT \\ \hline \downarrow \\ PL' \end{array} \left[\; COND \right. .$$

The reason the routine method is expressed as a transformation rule that syntactically substitutes the call to (ACHIEVE-FOR-ALL D G) is that in order to be restartable each loop has to have a global variable that contains the

```
(?TO-BE-EXPANDED
     == (ACHIEVE-FOR-ALL
           ?DESCR (λ (?GVAR) ?GOAL-EXP)))                |
─────────────────────────↓────────────────────────────── |
(LOOP                                                    |
     (PROCESS SINGLE-DELIVERY                            |
          (VALVE-REQUEST                                 |
              SINGLE-DELIVERY WHEELS '#F)                |  (EVAL (MAKE-RPL-VAR)
          (IF (NULL ?CURRENT-GOAL-ARG)                   |         ?CURRENT-GOAL-ARG))
          (!= ?CURRENT-GOAL-ARG                          |
               (PERCEIVE-1 (QUOTE ?DESCR))))             |
          (IF (NOT (NULL ?CURRENT-GOAL-ARG))             |
              (ACHIEVE ((λ (?GVAR) ?GOAL-EXP)            |
                           ?CURRENT-GOAL-ARG)))          |
          (VALVE-RELEASE                                 |
              SINGLE-DELIVERY WHEELS))                   |
 UNTIL (NULL ?CURRENT-GOAL-ARG)                          |
     (!= ?CURRENT-GOAL-ARG NIL))                         |
```

Fig. 4.10. Routine transformation rule for transforming a goal of the form (ACHIEVE-FOR-ALL *D* *G*) into a routine plan.

designator of the object for which the goal is to be accomplished. This variable has to be the same for a loop and the revisions of the loop but the variable must not occur in any other loop in the structured reactive plan. This is achieved by creating a global variable in the condition part of the routine transformation rule and using the variable created in the plan expansion.

4.4 Discussion

There are two important dimensions along which we should measure the success of the idea of having transparent structured reactive plans: (1) whether it is feasible to implement structured reactive plans such that they are transparent; and (2) whether the inferences and operations that we have listed in the beginning of the chapter are feasible and fast on transparent reactive plans. Evidence that the first claim is satisfied is given by the plan library we have implemented for the delivery robot and the design criteria for behavior-based systems (see chapter 2.2.8). We will show that transparent structured reactive plans are successful along the second dimensions in the chapters 6 and 8.

The idea of making programs more transparent to facilitate planning is not new. HACKER [156] uses annotations to make its plans more transparent. The following example shows an entry in HACKER's plan library for the task (ACHIEVE (ON ?A ?B)). Since the expressions bound to the second argument of the TO-predicate can be bound to complex pieces of code, HACKER makes its plans more transparent to reason about them. HACKER does so by adding indices or tags for pieces of the plan and making comments on the purpose of subplans (see figure 4.11 (left)). The code contains names for subprograms and lines are tagged with unique names. Comments about the plan

```
(TO (MAKE (ON ?A ?B))
     (HPROG MAKE-ON
               (LINE MO1 (PUTON ?A ?B))))

(GOAL MAKE-ON (MAKE (ON ?A ?B)))
(PURPOSE MO1 (MAKE (ON ?A ?B)) MAKE-ON)
  (TO (ACHIEVE (ON ?A ?B))
       (:TAG MAKE-ON
              (REDUCE (ACHIEVE (ON ?A ?B))
                 (WITH-POLICY (PROTECT ...)
                    (PLAN ((:TAG MO1
                                  (PUTON ?A ?B))))))))
```

Fig. 4.11. Entry in HACKER's plan library (left) and its reconstruction in RPL (right).

under construction are made in a data structure called "notebook." These comments describe the teleological structure of the plan using the GOAL- and PURPOSE-predicate. The GOAL-statement specifies that the subplan MAKE-ON is supposed to achieve (ON ?A ?B) in the final state of plan execution. The PURPOSE statement states that the purpose of the line MO1 is that (ON ?A ?B) holds at the end of MAKE-ON. This means that (ON ?A ?B) has to be protected from the end of MO1 to the end of MAKE-ON.

Figure 4.11 (right) shows how HACKER's entry in the plan library can be represented as a RPL plan that exhibits the same problem-solving behavior and makes its teleological structure explicit. Abstract goals are represented as "reducees," more detailed plans as "reducers," Sensitivity to protection violations is achieved by adding the statement (PROTECT ...) as a policy for the reducer. A top-level command (ACHIEVE *G*) has the same meaning as (GOAL *TAG G*).

GAPPS [101] and ERE [63] also have declarative achievement and maintenance statements for specifying top-level commands. In GAPPS these statements are used as abstract programming constructs. GAPPS compiles the declarative statements using a library of reduction rules into a synchronous circuit that maps an input vector into an output vector and checks some consistency conditions. ERE uses the declarative statements as behavior specifications that are checked against the projected behavior of the robot.

The notion of declarative statements in XFRM is more general. In XFRM, they are executable routine plans with a transparent structure, provide utility models and behavior specifications, and facilitate important inferences on structured reactive plans.

5. Representing Plan Revisions

Autonomous robots acting in changing and partly unknown environments, such as the delivery robot in the DeliveryWorld, often acquire important information while carrying out their plans. These robots cannot commit in advance to a fixed course of action; they have to be flexible and make critical decisions when they get the necessary information or when they cannot postpone the decisions any longer. As a consequence, their plans contain conditional, iterative, concurrent, and reactive subplans and are not just inflexible sequences of plan steps. To forestall flaws in the behaviors of robots with such flexible plans, planning systems do not only have to project how the robot will change the world as it executes the plan. They will also have to determine which parts of the plan were (projected to be) executed and why. In addition, they have to make their revisions in plans with a complex internal structure.

To make the implementation of such planning systems possible, we will focus in this chapter on another dimension of reasoning about plans and their projections: managing the complexity that is characteristic for the transformation of structured reactive plans. Over and above the difficult substantive issues about *what* plan transformations should be carried out, we must deal with formal issues about *how* to specify transformations on structured reactive plans. For example, the "protection-violation avoidance" transformation [156] is, in the classical framework, a matter of inserting a link. In a structured reactive plan, it is a complex editing operation that must respect the semantics of RPL. In the past [127] we have expressed such transformations as Lisp procedures, but the resulting codes are unintelligible and error-prone.

In this chapter I will describe XFRML (XFRM Language), the transformation language of the planning system XFRM. XFRML is a concise and declarative notation for implementing general, common-sense plan revision methods as plan transformation rules. The language is based on a PROLOG-like Horn clause language with fifty to sixty built-in predicates, most of which formalize relations on structured reactive plans and their execution scenarios. These built-in predicates constitute a layer of abstraction that presents structured reactive plans and their execution scenarios as if they were Horn clause databases. Making use of this abstraction, XFRM can retrieve information from, and test properties of, plans by means of concise, transparent queries that have a PROLOG-like semantics.

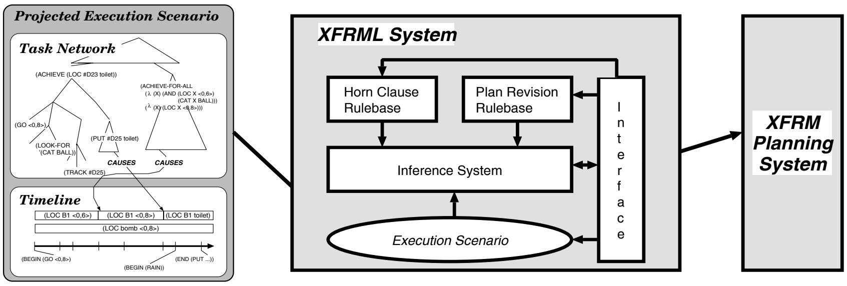


Fig. 5.1. The XFRML system.

The XFRML system consists of an inference system, a Horn clause database, and a transformation rule base. Using XFRML, XFRM reasons about and manipulate structured reactive plans by loading an execution scenario, asking queries in order to determine the behavior flaws in the scenario, diagnosing the causes of those flaws, and finally, applying plan transformation rules to produce better candidate plans.

Because of XFRML, XFRM can forestall execution failures caused by sensing routines that overlook or misperceive objects, by unfaithful tracking of objects through their descriptions, by unreliable physical actions with undesired effects, or by interference between multiple threads of the robot's own plan. These kinds of failures that are common in the execution of robot plans cannot be represented in other planning representations.

The design and implementation of XFRML contributes three important ideas to research in transformational planning of reactive behavior and notations for implementing planning modules for autonomous robots:

1. It shows that planning systems are able to anticipate, diagnose, and forestall a wide range of common flaws in the behavior of autonomous robots.
2. A set of fifty to sixty built-in predicates is sufficient to describe the important aspects of the structure of structured reactive plans: the physical effects of plan execution, the process of plan interpretation, and temporal, causal, and teleological relationships.
3. The built-in predicates can be implemented to be fast, if the structured reactive plans are *transparent.*[1]

[1] We have defined the transparency of a structured reactive plan (see chapter 4.2) as follows: A plan is transparent with respect to achieving a state G if the plan satisfies the following two conditions: (1) For each call of a control routine C that has an intended effect G, there exists a plan P of the form (REDUCE (ACHIEVE G) $BODY$) such that $BODY$ contains C and (2) For each plan of the form (REDUCE (ACHIEVE G) $BODY$), $BODY$ is locally relevant for achieving G. A plan is locally relevant for achieving a goal G if there exists a situation in which $BODY$ has a high probability of achieving G.

This chapter is organized into three parts. The first part introduces the concepts and objects XFRML reasons about and manipulates. The second part lists and describes the important predicates of XFRML. The last part describes the implementation of plan transformation rules. We conclude with a discussion of XFRML.

5.1 Conceptualization

In XFRML, plan revisions are formalized as plan-transformation rules, which have two components: an *if* and a *then* part. The *then* part spells out what plan editing to do, based on tests (in the *if* part) of what the planner predicts might happen if the current plan were executed. What might happen is determined by projecting the plan and represented as an execution scenario (see chapter 3.2.2).

The predicates for accessing planner data structures form the interesting part of XFRML. Let's start by looking at some of XFRM's plan-transformation rules we have implemented in XFRML:

1. **IF** a goal might be clobbered by an exogenous event, **THEN** stabilize the goal state immediately after achieving it.
2. **IF** a goal might be clobbered by a robot action, **THEN** make sure the clobbering subplan be executed before the clobbered goal is achieved.
3. **IF** a goal might be clobbered by a robot action and the robot knows it (for example, the designator describing the object for which the goal is to be achieved specifies that the goal does not hold for the object), **THEN** reachieve the clobbered goal as the very last step in your plan.
4. **IF** a goal might be left unachieved because the robot overlooked an object, **THEN** use another sensing routine for perceiving this object.
5. **IF** a goal might be left unachieved because the robot had an ambiguous object description, **THEN** achieve the goal for all objects satisfying the object description.
6. **IF** the robot might erroneously achieve a goal *G* for an object *OB'* instead of the intended object *OB* because the robot changed the world earlier, such that its description for *OB* became ambiguous, **THEN** perform the subplan that caused the ambiguity after goal *G* is achieved.

Since the robot's world model is incomplete, probabilistic, and even faulty, XFRM can only guess what *might* happen when the robot will execute a plan. To do so, XFRM generates several random execution scenarios by projecting the plan and assumes that something might happen, if it happened in at least one of those scenarios. Thus, a plan revision is applicable if its applicability condition is satisfied by one of these execution scenarios.

Testing the applicability of the failure-type-specific plan revisions listed above requires the planning system to diagnose projected behavior flaws. In the diagnosis process the planning system must be able to carry out at least the following inferences:

1. Infer what the outcomes of subplans were and what they should have been;
2. Infer which parts of the plan terminated abnormally; and
3. Reconstruct the computational state of plan execution that influenced a control decision.

The repair processes required by these six example revisions must carry out operations such as

1. adding steps to the plan;
2. reordering parts of the plan;
3. introducing repetition of steps until some condition is satisfied; and
4. changing operations on a single object into iterated operations on every element of a set of objects.

Other revision methods remove steps from a plan, change variable bindings and arguments to routines, and introduce a more robust control structure. In general, plan modifications in XFRML are arbitrary transformations of pieces of RPL plans.

To get a sense of what entities and relations to include in a conceptualization of execution scenarios, we analyze a typical plan revision we would like XFRM to perform (number 1 above). Restating the revision method more carefully leads us to the following plan revision rule:

$$\frac{P_{ACHIEVE(GOAL)}}{\text{SEQ } P_{ACHIEVE(GOAL)} \ STABILIZE\,(GOAL(VAR))}\downarrow$$

1. *GOAL(OB)* is clobbered by an exogenous event;
2. *DESIG* is a data structure returned by the sensing routine when the routine saw *OB*;
3. the robot tried to achieve *GOAL(DESIG)* with a plan named $P_{ACHIEVE(GOAL)}$; and
4. there exists a RPL variable VAR with value *DESIG*,

which can be read procedurally as

```
IF 1. GOAL(OB) is clobbered by an exogenous event;
   2. DESIG is a data structure returned by the sensing routine
      when the routine saw OB;
   3. the robot tried to achieve GOAL(DESIG) with a plan
      named $P_{ACHIEVE(GOAL)}$; and
   4. there exists a RPL variable VAR with value DESIG;
THEN REPLACE $P_{ACHIEVE(GOAL)}$
     WITH    SEQ $P_{ACHIEVE(GOAL)}$
                 STABILIZE(GOAL(VAR)).
```

The conditions of this plan revision rule refer to different aspects of plan execution: the first condition specifies what happened in the world, namely that a goal was destroyed by an event not under the control of the robot. The second condition relates a data structure used in the plan with the object it designates. The data structure *DESIG* is the computational result of sensing object *OB*. The third condition checks whether the robot controller has executed a subplan with the purpose to achieve *GOAL(DESIG)*. The last condition tests the existence of a variable in the reactive plan that has a description of the object *OB* as its value.

This plan revision rule might seem more complicated than it needs to be. It is not. Most planning systems use simpler representations. They either reason about how the world changes when the plan gets executed [7] or about the process of plan interpretation [52, 122]. In the first case, the planning systems have to assume that the critical decisions about the course of actions are made during planning because they cannot represent and rationalize decisions made during plan execution. In the second case, they have to assume that the physical effects of the interpretation of a plan piece are exactly known in advance because they cannot predict how the world changes.

For most autonomous robots as well as for the delivery robot in the DeliveryWorld we do not want to make these assumptions because many flaws in their behavior are caused by the state of plan interpretation not reflecting the state of the world: the interpretation of the grasp routine is completed but the object is not in the robot's hand, the robot has overlooked an object, and so on. To diagnose these kinds of behavior flaws we need more complex representations that represent the physical effects of plan execution, the process of plan interpretation, and the relationship between both.

For example, because structured reactive controllers do not assume that arguments of plan steps denotate objects in the world, they cannot confine themselves to representing just the "object." Instead, planning processes in structured reactive controllers need to reason about three closely related concepts: the *object* itself, the *object description*, which is generated by sensor routines and used for plan interpretation, and the RPL *variable*, which is the syntactic object in an RPL plan that, during plan interpretation, contains the object description as its value. Thus, to revise the plan such that the robot will make sure that BALL-37 will not disappear after its delivery, the planner has to infer which designator *DES* in the process of plan interpretation denotes BALL-37 and which variable *VAR* has the value *DES* to make the appropriate plan transformation.

To provide the information necessary to test the applicability conditions of transformation rules like the one above, our conceptualization of an execution scenario consists of a *PLAN*, a *timeline*, and a *task network.* The *plan* was projected in order to produce the execution scenario. The *timeline* represents how the world changes, and the *task network* represents how the computational state of the reactive plan changes as the plan gets executed.

We will start our description of components of the execution scenarios by detailing the task network and the timeline. When the plan is executed, the plan interpreter generates a structure of stack frames, called the *task network*. A task corresponds to the execution of a node in the code tree. When the interpreter is run in projection mode, the stack frames are preserved for later inspection by plan transformations. The task network stores the necessary information to decide whether a given task has been tried, whether the task signalled a failure, and what value it has returned. If a planner analyzes a plan failure in a plan in which the execution depends on the plan's internal state, the planner will have to reconstruct the value of variables and data structures the plan is using at any time during projection. The necessary information is stored in the data structure *task effects*. The if-then rules need to be able to access the task network (but fortunately not change it). The task network enables the planner to reconstruct the precise state of the robot controller before and after any task.

Transformation rules need to access another data structure: the timeline the projector uses to represent what happens in the world as the plan is executed. This is not a tree but a linear sequence of events and their results, synchronized with the task network. The timeline is a sequence of time instants where each time instant is the start or end of a physical or sensing action of the robot, or an external event. With each time instant there is associated a set of *occasions* and a set of *events*. Occasions are ground atomic statements that describe the state of the effectors and sensors, sensor data, and the environment. The events represent what happened at this time instant.

The final component of an execution scenario is the code tree of the projected plan. For carrying out plan revisions, structured reactive plans are best thought of as code trees and their revision as replacing subtrees in code trees. Therefore, in an execution scenario, the projected plan is represented as a structure, called a *code tree*, which essentially duplicates the structure of the plan text, augmented with some further syntactic information. (For details, see [127].) The *if* parts of transformation rules need to access that tree, and the *then* parts need to rearrange it.

Figure 5.2 shows how a XFRML plan revision rule edits the code tree of a RPL plan. The plan revision identifies the subplan *P* which requires an object

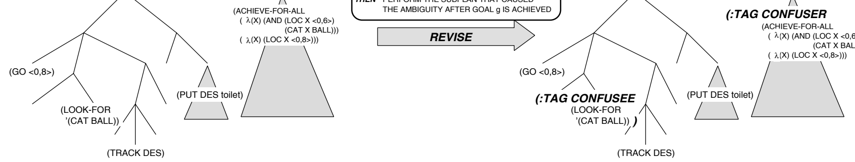

Code Tree
(ACHIEVE ((λ(D) (LOC D toilet)) DES)
(ACHIEVE-FOR-ALL (λ(X) (AND (LOC X <0,6>) (CAT X BALL))) (λ(X) (LOC X <0,8>)))
(GO <0,8>)
(LOOK-FOR '(CAT BALL))
(TRACK DES)
(PUT DES toilet)
IF A GOAL g WAS ACHIEVED FOR THE WRONG OBJECT BECAUSE THE ROBOT MADE AN OBJECT DESCRIPTION AMBIGUOUS
THEN PERFORM THE SUBPLAN THAT CAUSED THE AMBIGUITY AFTER GOAL g IS ACHIEVED
REVISE
& (BEFORE CONFUSEE CONFUSER)
ACHIEVE ((λ(D) (LOC D toilet)) DES)
(:TAG CONFUSER (ACHIEVE-FOR-ALL (λ(X) (AND (LOC X <0,6>) (CAT X BALL))) (λ(X) (LOC X <0,8>))))
(GO <0,8>)
(:TAG CONFUSEE (LOOK-FOR '(CAT BALL)))
(PUT DES toilet)
(TRACK DES)

Fig. 5.2. Plan revision rule transforming a code tree.

description to be unambiguous, the subplan *C* which causes the ambiguity, and a subplan *PO* that contains *P* and *C* and allows constraining the order of its subplans, i.e., has the form (PARTIAL-ORDER *STEPS CONSTRAINTS*). Note, the plan revision requires a notation for plan revision to identify subplans by the states they cause, their purpose, and their form. The revision makes three changes in the code tree. It names the subplans *P* and *C* by replacing *P* by (:TAG *CONFUSEE P*) and *C* by (:TAG *CONFUSER C*). The third change adds (:ORDER *CONFUSEE CONFUSER*) to the constraints of the subplan *PO*.

5.2 Making Inferences about Execution Scenarios

We have seen that the transformation rules listed in the previous section require a planning system to navigate within and between task networks, timelines and code trees in order to extract the necessary information from execution scenarios. The purpose of XFRML is to simplify the retrieval of this information and the manipulation of plans by abstracting away from the complex implementation of execution scenarios. Using XFRML queries, the planning system can specify what information it needs and the built-in predicates retrieve the information automatically.

Conceptually, an execution scenario consists of the plan, the plan interpretation, and the world, which are described in the sections 5.2.2, 5.2.3, and 5.2.4 respectively. The following table lists the most important concepts in the conceptualization of execution scenarios.

plan	**plan interpretation**	**world**
plan	task	event
RPL variable	designator	object
fluent	RPL value	state
code tree	state of plan interpretation	time instant
plan text	status	occasion
top-level plan	return value	
ordering context		

As important as the concepts themselves are the relations among some of them.

plan		**plan interpretation**		**world**
plan	↔	task	↔	event
RPL variable	↔	designator	↔	object
fluent	↔	RPL value	↔	state/event
		state of plan interpretation	↔	time instant

Let us consider the relations between plans, tasks, and events. The interpretation of plans (represented by tasks) changes the computational state of the structured reactive controller, and the changes of the computational state, in particular the activation and deactivation of continuous control processes cause events (changes) in the world. Thus, if a goal is clobbered by an event that is an action of the robot then the planning process might have to constrain the plan such that the interpretation of the subplan that achieves the goal does not start before the interpretation of the subplan that caused the clobbering event is completed.

Another important relationship is the one between fluents, values, and states/events. Fluents are sometimes directly connected to the sensors of the robot to detect particular events or states in the world. For example, the fluent HAND-FORCE-SENSOR is connected to the force sensor in the robot's hand and can be used to detect when an object slips out of the robot's hand.

We will discuss the conceptualization of these relations between plans, plan interpretation, and the state of the world in section 5.2.5.

The syntax of XFRML is the usual PROLOG-in-LISP combination: parenthesized prefix notation, with variables indicated with "?" and segment variables by "!?". There are some very general predicates that provide basic utilities (arithmetic, unification) and the ability to call LISP code. The predicate (== E_1 E_2) succeeds if E_1 unifies with E_2. (SETOF *ARG S SET*) succeeds if *SET* is the set of all *ARG*s that satisfy *S*. And so on. The two predicates EVAL and LISP-PRED constitute the LISP interface of XFRML. The predicate (EVAL *EXP VAL*) is true if *VAL* is the value returned by evaluating *EXP*. The predicate (LISP-PRED *PRED* $X_1 \ldots X_n$) succeeds if evaluating *(PRED* $X_1 \ldots X_n$*)* with the LISP interpreter returns a "non-nil" value.

XFRML allows us to define new predicates by a set of Horn clause rules of the form: ($\leftarrow$ P (and Q_1 ... Q_n)) where P is an atomic formula and Q_1, ..., Q_n literals. Negation is implemented as negation by failure. For instance, a programmer can define the predicate (MEMBER *E L)* that holds if E is an element in the list L, as follows:

```
(DEF-FACT-GROUP MEMBER

  (← (MEMBER ?E ())
     (FALSE))

  (← (MEMBER ?HEAD (?HEAD !?TAIL))
     (TRUE))

  (← (MEMBER ?E (?HEAD !?TAIL))
     (MEMBER ?E ?TAIL))).
```

The first Horn clause in the fact group specifies that there exists no object that is a member of the empty list. The second rule specifies that an object is a member of a list if it unifies the first element in the list. The third clause specifies that an object is a member of a list if it is a member of the tail of the list.

Queries have the form (?- Q) where Q is a literal or a conjunction of literals. Thus, given the query (?- (MEMBER 1 (1 2 3))), the XFRML system that contains the definition of MEMBER in its rulebase would answer TRUE; given the query (?- (MEMBER ?X (1 2 3))), it would answer (TRUE (((X 1)) ((X 2)) ((X 3)))), meaning that the query is satisfied if the system substitutes 1, 2, or 3 for the variable ?X. (?-VAR ?X (MEMBER ?X (1 2 3))) returns (1 2 3), all bindings of ?X that make the query true.

5.2.1 Some Examples

Before we introduce and describe the built-in predicates, namely those that we use to formulate queries about execution scenarios in order to detect and diagnose plan failures, we will first give some examples of tests that have occurred in actual rules.

- The robot tried to get a ball from location ⟨*0,8*⟩ into its hand:

```
(AND (TASK-GOAL ?TSK (ACHIEVE (LOC ?DES HAND)))
     (HOLDS (VISUAL-TRACK ?DES ?OB) (DURING ?TSK))
     (HOLDS (AND (CATEGORY ?OB BALL) (LOC ?OB ⟨0,8⟩))
            (BEGIN-TASK TOP))).
```

  In more detail, the robot has carried out a task ?TSK of the form (ACHIEVE (LOC ?DES *HAND*)) for a designator *DES*. While carrying out ?TSK the robot was for some time visually tracking the object described by *DES* by pointing the camera at the object ?OB where ?OB is a ball at location ⟨*0,8*⟩.
- The robot overlooked a ball at ⟨*0,8*⟩, that is, the routine for finding a ball at location ⟨*0,8*⟩ did not return an object description (RETURNED-VALUE ?P-TSK {}) although there was a ball at ⟨*0,8*⟩ (HOLDS (AND (CATEGORY ?X BALL) (LOC ?X ⟨*0,8*⟩)) (TT ?P-TSK)).

```
(AND (TASK-GOAL ?P-TSK
                (PERCEIVE-1
                  (λ (?X) (AND (CATEGORY ?X BALL)
                               (LOC ?X ⟨0,8⟩)))))
     (RETURNED-VALUE ?P-TSK {})
     (HOLDS (AND (CATEGORY ?X BALL) (LOC ?X ⟨0,8⟩))
            (TT ?P-TSK)))
```

- After the execution of plan P the robot "knows" it has a ball from location ⟨*0,8*⟩ in its hand.

```
(AND (TASK-GOAL ?A-TSK (ACHIEVE (LOC ?DES HAND)))
     (HOLDS (TRACK ?DES ?OB) (DURING ?A-TSK))
     (HOLDS (AND (CATEGORY ?OB BALL) (LOC ?OB ⟨0,8⟩))
            (BEGIN-TASK TOP))
     (BELIEF-AT (LOC ?DES HAND) (END-TASK TOP)))
```

5.2.2 Accessing Code Trees

A basic requirement for our notation is that we will be able to identify parts of the plan, that is, parts of the code tree.

XFRML distinguishes between the following categories of objects for plan representation: RPL expressions, RPL code, and code paths. RPL expressions are S-expressions that represent the plan text. For the following discussion we will use a subplan *P* that is part of the plan described on page 62:

```
(SEQ (GO 2 2)
     (!= OBS (LOOK-FOR ((CATEGORY BALL)) 0))
     (IF (NOT (NULL OBS))
         (LET ((OB (CAR OBS)))
             (START-TRACKING OB 0))
        (FAIL))).
```

This form of plan representation is convenient for pattern matching. For instance, a planning effort can match the plan *P* with a particular plan pattern (SEQ (?PROC-NAME !?ARGS) !?STEPS) using the predicate == that unifies two XFRML terms. Thus the XFRML query (?- (== *P* (SEQ (?PROC-NAME !?ARGS) !?STEPS))) returns the result:

```
(TRUE (((?PROC-NAME   GO)
        (?ARGS        (2 2))
        (?STEPS       ((!= OBS (LOOK-FOR ((CATEGORY BALL)) 0))
                       (IF (NOT (NULL OBS))
                           (LET ((OB (CAR OBS)))
                               (START-TRACKING OB 0))
                          (FAIL))))))).
```

RPL code is a plan representation that captures aspects of the structure of RPL plans better: the location of code pieces in the code tree and the subcode relationship. The code path is a representation of the location of a piece of RPL code in the code tree of a structured reactive plan.

Indexing Subplans with Tasks. Every task in the task network is generated by the interpretation of a piece of the code tree. The predicate (TASK-CODE-PATH *TSK CP*) will hold if *CP* is the "codepath" of the subplan executed in order to perform *TSK*. If the plan is not fleshed out enough to contain the subplan, the predicate will not hold. A *codepath* specifies a subtree of a code tree by giving the labels of the branches that lead to it (see chapter 3.1.1).

Of course, to use TASK-CODE-PATH, we must have a task. We have spelled out several ways to find tasks in the task network on page 107. One possibility that we mention here is to use the relation (TASK-GOAL *TSK GOAL*) which holds if *TSK* is a task with the purpose to achieve *GOAL*. The relation TASK-GOAL can be used to test whether the robot ever tried to get something to location $\langle 0,10 \rangle$:

(TASK-GOAL ?TSK (ACHIEVE (LOC ?DES $\langle 0,10 \rangle$))).

We will discuss on page 113 how to test whether the goal was to take a *particular* object to $\langle 0,10 \rangle$.

The Top-level Plan. Another way to find subplans is to exploit the stereotyped form of the top-level plan.

```
(WITH-POLICY P1
  ...
    (WITH-POLICY Pk
        (PARTIAL-ORDER
            ((:TAG MAIN (TOP-LEVEL
                                        (:TAG TLC1 CMD1)
                                        ...
                                        (:TAG TLCn CMDn)))
              —OTHER-STEPS—)
          —ORDERINGS—))))
```

The TOP-LEVEL action is a set of tagged commands corresponding to the user's instructions. The plan TOP-LEVEL is a step in a PARTIAL-ORDER statement that can contain other steps and ordering constraints on plan steps. Wrapped around the main partial order statement is a set of policies, constraints on the execution of the plan. Such policies are, for instance, *"avoid obstacles"* or *"don't run out of gas"*. (MAIN-PARTIAL-ORDER (PARTIAL-ORDER *STEPS ORDERINGS*) *CODE-PATH*) is the XFRML predicate that identifies these pieces. Common revisions of the top-level plan are the addition of steps to the beginning and end of a plan, or the addition of ordering constraints on the top-level tasks or their subtasks. To get an ignition key in the beginning the planner adds a step (:TAG GET-KEY (GET-IGNITION-KEY)) to *—OTHER-STEPS—* and (:ORDER GET-KEY MAIN) to *—ORDERINGS—*.

Preparing Ordering Constraints. A common plan revision is to add ordering relations between subplans such that a particular task T_1 will be executed necessarily before another task T_2. However, we cannot always directly reorder two tasks of interest. Consider the following piece of RPL code (PAR (LOOP (FOO) UNTIL (BAR)) (BAZ)) and assume the execution of (FOO) in the third iteration of (LOOP (FOO) UNTIL (BAR)) clobbers a goal achieved by (BAZ). Hence we want to make sure that (FOO) will come first. Because RPL has no mechanism for referring to particular iterations of a task, and because we usually suspect there's nothing special about the third iteration, we have to order the entire LOOP before (BAZ). In general, for any two tasks, T_1, T_2, we seek three tasks: a supertask of both of them where we can store the ordering information (call this their "partial-order locus"); the outermost supertask of T_1 below the partial-order locus that can be properly referred to (call this T_1's "safe supertask"); and the analogous "safe supertask" of T_2. This pattern occurs repeatedly in XFRM transformations, and we capture it with a built-in predicate ORDERING-CONTEXT:

(ORDERING-CONTEXT T_1 T_2 $SAFE\text{-}SUPER_1$ $SAFE\text{-}SUPER_2$ *P-O-LOCUS*).

5.2.3 Predicates on Plan Interpretations

In this section, we discuss predicates for referring to tasks, and, at a finer level, to the computational objects that were built and used when tasks were executed.

Conceptualization of Plan Interpretations. A projected plan interpretation is represented as a sequence of states of plan interpretation. A state of plan interpretation is the start or end of a task. The states of plan interpretation are enumerated by the two functions (BEGIN-TASK *TSK*) and (END-TASK *TSK*). Thus for each task *TSK* there exists exactly one state of plan interpretation (BEGIN-TASK *TSK*) at which the interpretation of *TSK* starts and one (END-TASK *TSK*) at which the interpretation of *TSK* terminates.

A state of plan interpretation consists of the tasks that exist in this state and their computational states. The computational states of tasks are characterized by the task status and the task environment. The status of tasks can be either created, active, done, failed, or evaporated. The task environment maps the variables and fluents in the scope of the task to their respective values. The existing processes are characterized by their status and the valves they own.

Thus the concepts that describe a state of plan interpretation are: tasks, task status, RPL variables and fluents, and the values of variables and fluents. The events that transform a state of plan interpretation into another one are the start or end of the interpretation of a task.

Predicates on Plan Interpretations. As we said above, tasks are like stack frames: they record how the code subtree associated with the task was interpreted. We use the relation (TASK *TSK*) to state that *TSK* is a task in the given projection, and (TASK-PLAN *TSK CODE*) to state that *TSK* is the task to execute subcode *CODE*. The description of a complete plan interpretation comprises a *task network*, a graph where the nodes are tasks and the arcs represent the "subtask" and temporal orderings between tasks. The relation (SUBTASK TSK_1 TSK_2) states that TSK_1 is a subtask of TSK_2. SUBTASK+ is the transitive closure of SUBTASK.

When the plan interpreter starts interpreting the plan P_{TSK} of task *TSK* the interpreter changes the status of *TSK* from *created* to *active* and creates a state of plan interpretation (BEGIN-TASK *TSK*). (TASK-STATUS *TSK* $\left\{ \begin{matrix} \textit{(BEGIN-TASK TSK')} \\ \textit{(END-TASK TSK')} \end{matrix} \right\}$ *STATUS*) holds if *STATUS* is the status of *TSK* at the start and end of *TSK'*. A task can come to an end in three different ways. It can *fail*, that is, a failure can be reported during the interpretation of P_{TSK} from which P_{TSK} cannot recover. It can *succeed*, i.e., P_{TSK} terminates without reporting a failure. Finally, it can *evaporate* if the task became pointless for outside reasons. In these cases the task status changes from *active* to *failed*, *succeeded*, or *evaporated* respectively. (TASK-OUTCOME *TSK STATUS*) is the status of *TSK* after *TSK* is ended, that is succeeded, failed, or evaporated. In each of these cases a new state of plan interpretation, *(END-TASK TSK)* is generated. Tasks may also return values when they succeed. For instance, sensing tasks return data structures describing objects they have found in an image. The relation (RETURNED-VALUE *TSK VAL*) holds if *VAL* is the value returned by *TSK*.

Variables in RPL plans are used for storing information about the world, doing simple program control (e.g., counters), and communication between different processes in a reactive plan. Some variables are *fluents*, registers that can be set by primitive behaviors. For instance, the fluent ODOMETER* is continually updated by the behavior moving the robot. Variables can be global, that is, accessed from any part of the plan, or local — visible only within a limited scope in the plan. The values of variables usually change during plan interpretation. The relation (VALUE *VAR* $\left\{ \begin{array}{c} \textit{(BEGIN-TASK TSK)} \\ \textit{(END-TASK TSK)} \end{array} \right\}$ *VAL*) is true if *TSK* is in the scope of the plan variable *VAR* and *VAR* has the value *VAL* at the start or end of *TSK*.

Predicates on Top-level Commands. Top-level commands have a particular meaning in RPL plans. They specify the objectives that the robot has to accomplish. XFRML provides two predicates that specify relations on top-level commands. The first one is (TLC-TASK *TLC TSK*) that holds if *TSK* is the task in the interpretation of a structured reactive controller with the purpose to accomplish the top-level command *TLC*. The second relation (TLC-NAME *TLC NAME*) holds if *NAME* is the name of the top-level command *TLC*. Using the predicate TLC-TASK, the planner can issue the following XFRML query in order to determine the top-level commands for which the structured reactive controller has signalled an irrecoverable execution failure.

```
(?-VAR ?CMDS (SETOF ?TLC-CMD
                    (AND (TLC-TASK ?TLC-CMD ?TLC-TASK)
                         (TASK-OUTCOME ?TLC-TASK FAIL))
                    ?CMDS))
```

5.2.4 Predicates on Timelines

The purpose of plans is to change the world. XFRM represents how the world changes during the execution of a plan in terms of *time instants*, *occasions*, and *events* [128]. *Time instants* are points in time at which the world changes due to an action of the robot or an exogenous event. Each time instant has a *date* which holds the time at that particular time instant occurred. An *occasion* is a stretch of time over which a world state *S* holds and is specified by a proposition, which describes *S*, and the time interval for which proposition is true. More than one event can occur at a time instant, but only one of them can effect changes in the world. We will call this the *causing event* and the set of all occurring events the *happenings* of the time instant.

There are three types of XFRML relations that describe the projected state of the environment and how and why it changes:

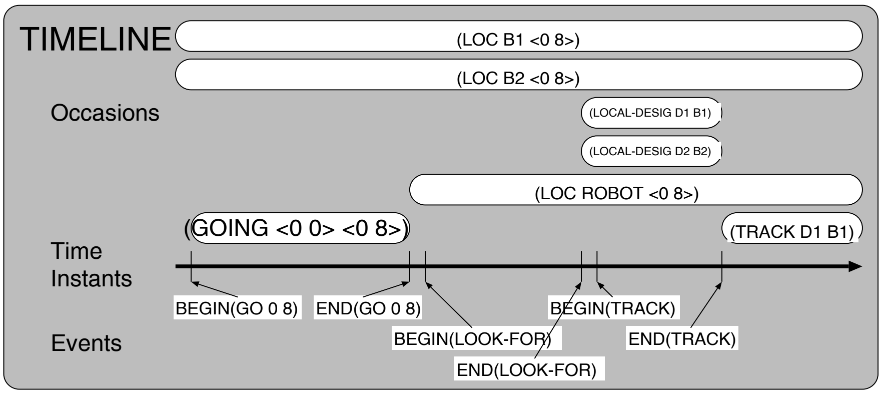


Fig. 5.3. Graphical representation of a timeline.

1. **The HOLDS-predicate.** Relations on timelines characterize the truth of a world state at a given time. The relation (HOLDS *STATE T*) is true if *STATE* holds for *T* where *STATE* is a conjunction of literals and *T* is the start or end of a task, the interval over which a task was active. If *T* has the form (TT *TSK*), then *STATE* has to hold throughout the time interval starting at (BEGIN-TASK *TSK*) and ending at (END-TASK *TSK*). It it has the form (DURING *TSK*) it has to hold during a subinterval thereof.
2. **Representing Change.** Besides the truth of states for time instants or intervals, XFRML provides relations describing change and its cause. The relation (CLOBBERS *EV STATE TI*) holds if *EV* is the last event occurring prior to *TI* that causes *STATE* to be false. Similarly, (CAUSES *EV STATE TI*) holds if *EV* is the last event occurring prior to *TI* that causes *STATE* to hold.
3. **Representing Different Types of Events.** We distinguish between two classes of events, events not under the robot's control, (EXOGENOUS-EVENT *EV*), and events caused by carrying out a task, (ACTION *EV TSK*).

5.2.5 Timelines and Plan Interpretation

The fourth component of our conceptualization describes how plan interpretation and change in the world are related. An important class of relations between plan interpretation and the physical world are causal: reactive plans start primitive control routines *(behaviors)* and thereby change the world. Plans also ask the robot to sense its surroundings and thereby change its own computational state of plans by storing the results of sensing operations. The relation (ACTION *EV TSK*) holds if *EV* is an event caused by a robot action part of task *TSK*. A second class of relations is teleological: the purpose of some subplans/tasks is to achieve, maintain, perceive, and stabilize a particular world state. A third relationship is representational: data structures in reactive plans are often used to (partially) describe objects and states in the world. The relation (AT-TIMEINSTANT *SOPI TI*) states that the state of plan interpretation *SOPI* occurs roughly at time instant *TI*.

Teleological Relations. The purpose of many subplans can be described in terms of world states. We assume that if a plan is intended to achieve, maintain, or perceive a particular state, then this purpose is represented explicitly (see chapter 4). We represent a plan P for getting an object described by the data structure *DES* to the location $\langle x,y \rangle$ as (REDUCE (ACHIEVE (LOC *DES* ⟨*0,10*⟩)) *P*) [19]. When the plan interpreter executes this statement it generates two subtasks, T_A for (ACHIEVE (LOC *DES* ⟨*0,10*⟩)) and T_R for P, and performs T_A by performing T_R. The planner views T_A as an annotation for T_R that represents the purpose of T_R [156].

We promised on page 109 to show how to test whether a task's purpose was to get a particular object, say *BALL-5*, to location ⟨*0,10*⟩. Here's how:

```
(AND (TASK-GOAL ?TSK (ACHIEVE (LOC ?DES ⟨0,10⟩)))
     (HOLDS (TRACK ?DES BALL-5) (DURING ?TSK))).
```

The first conjunct is satisfied if there exists a task of the form (ACHIEVE (LOC ?DES ⟨*0,10*⟩)). Since the first argument of the goal, ?DES, is a data structure describing an object and not the object itself, we check in the second condition that ?DES corresponds to the object *BALL-5*. One way to establish this correspondence is to check the condition (TRACK ?DES *BALL-5*), which is true if the robot tracked ?DES by pointing the camera at *BALL-5*.[2]

Representational Relations. Most variables and data structures in a structured reactive plan serve as internal models of aspects of the world. For instance, the registers X-POS and Y-POS, continually updated by the robot's odometer as the robot moves, provide the robot with an estimation of its own position in the world. In this case we say that at any time the robot believes it is at location $\langle X, Y \rangle$ if in the current computational state, the value of X-POS is X and the value of Y-POS is Y (see [142]). We take the relation (BELIEF-AT *STATE* $\left\{ \begin{array}{c} \textit{(BEGIN-TASK TSK)} \\ \textit{(END-TASK TSK)} \end{array} \right\}$) to mean that the robot believes *STATE* holds in the world.

The programmer specifies axioms that define under which conditions the robot believes in *STATE*. Such axioms all have the same form: they require that the value of global variables satisfy some constraints. For example, she can axiomatize the meaning of X-POS and Y-POS using the following rule:

```
(BELIEF-AT (LOC ROBOT ⟨?X,?Y⟩) ?AT)
  ←  (VALUE X-POS ?AT ?X)
     (VALUE Y-POS ?AT ?Y),
```

[2] Formally, (TRACK ?DES *BALL-5*) is a function that maps ?DES and *BALL-5* into the set of time instants in which the robot tracked ?DES by pointing the camera at *BALL-5*. (DURING ?TSK) should be (TIME-INSTANTS ?TSK) a function that maps ?TSK into the sets of time instants that happen between the begin and the end of ?TSK. HOLDS should be (HOLDS-DURING tis1 tis2), which is true if the intersection between tis1 and tis2 is not empty.

which says at timeinstant ?AT the robot will believe to be at location ⟨*?X, ?Y*⟩, if at ?AT the value of the global variable X-POS is ?X and the value of Y-POS is ?Y. Using this rule, XFRM can check whether a possible cause for an execution failure is an incorrect odometer reading by asking the following XFRML query:

```
(AND (BELIEF-AT (LOC ROBOT ⟨?X,?Y⟩) (END-TASK TSK))
     (THNOT (HOLDS (LOC ROBOT ⟨?X,?Y⟩)
                   (END-TASK TSK))).
```

Causal Relations. The causal relations are those that connect physical actions, exogenous events, and computational actions such as the interpretation of tasks to changes in the environment. (CAUSING-EVENT *TI EV*) holds if *EV* is the event that caused the effects beginning at the time instant *TI*. *EV* can be either an exogenous event or a robot action. If *EV* is a robot action the programmer can also ask the query (CAUSING-TASK *TI TSK*). This query determines the smallest task that caused the robot action and thereby the state transition to *TI*.

The programmer can also ask how the world changed during the interpretation of a task by asking queries using the HOLDS predicate. Such a query determines which states held in the beginning and end of the task, for some time during the task, and throughout the task.

Sometimes subplans are diagnosed to fail because a particular state in the world does or does not hold. If this is the case, it is important to find what made the state true or false. This information can be inferred from an execution scenario using the queries (CLOBBERS-BEFORE *EV STATE* (END-TASK *TSK*)) and (CAUSES-BEFORE *EV STATE* (END-TASK *TSK*)).

Temporal Relations. In execution scenarios time is gauged with two different measures: the time passed in the global clock and the change of states of plan interpretation. Often in the course of diagnosing a plan failure it is necessary to retrieve the states of plan interpretations that occurred at a particular time instant or to find the time instant that corresponds to a particular state of interpretation. Since plan interpretation is very fast compared to the time it takes to perform a physical action, the robot controller avoids performing expensive computations or at least runs expensive computations in parallel such that the robot actions do not have to wait for those computations to be completed. The relation (OCCURS (BEGIN-TASK *TSK*) *TI*) makes this connection between time instants and states of plan interpretation. (OCCURS (BEGIN-TASK *TSK*) *TI*) holds if the state of plan interpretation occurs approximately at the moment of time instant *TI*.

5.3 Expressing Plan Revisions

So far we have focused on how inferences are made about planner data structures. Now we turn our attention to how the results of these inferences are used to revise code trees.

We will diagram plan-transformation rules

```
PL AT CP == PAT ┌
                │
──────↓──────   │ COND
                │
      PL'       └
```

where *cond* is the applicability condition, *PL* AT *CP* == *PAT* the input plan schema, and *pl'* the output plan schema of the rule. The applicability condition is a conjunction of XFRML clauses. The input plan schema consists of a pattern variable to which the subplan with code path *CP* is bound and a pattern *PAT* that is matched against the subplan *PL*. The pattern is optional. *pl'* is an RPL plan fragment containing pattern variables that get instantiated using the bindings produced by proving the applicability condition and matching the input plan schema. The rule is applicable if the application condition holds and the plan *PL* with code path *CP* matches the pattern *PAT*. The resulting plan fragment replaces the input fragment in the revised plan.

Figure 5.4 shows the formalization of the first plan revision method discussed in section 5.1. Let ?UNACH-GOAL be bound to the goal that is not achieved at the end of the task ?TSK, the task to execute the top-level plan. The revision replaces a plan step ?TO-BE-REVISED with a sequence (SEQ) consisting of ?TO-BE-REVISED, followed by a step stabilizing the previously achieved goal; it does this under a set of conditions specifying that (a) the unachieved subgoal is of the form (?PRED ?OB ?VAL); (b) the robot tried to achieve a goal (ACHIEVE (?PRED ?DES ?VAL)) as a subtask of ?TSK (c); (d) during this subtask the robot tracked ?DES by pointing the camera at object ?OB; (e) the unachieved goal holds at the end of subtask ?SUBTSK; (f) ?VAR is a variable in the environment of the subplan that has the value ?DES at the begin of task ?SUBTSK; and (g) the ?CP is the code path of ?SUBTSK.

Many plan revisions require the modification of more than one subplan. For instance, the following plan revision introduces an ordering constraint between two subplans that were predicted to have possibly harmful interferences. For this revision, let ?VIOLATOR be a task that clobbers a goal achieved by the task ?PROTECTEE. The application condition computes the ordering context for these two tasks and generates two labels (tags) for plan steps that can be used in ordering constraints. The three modifications performed by this plan revision are: tagging both safe supertasks of the VIOLA-

```
                                 ┌ (AND (== ?UNACH-GOAL (?PRED ?OB ?VAL))              (a)
 (?TO-BE-REVISED AT ?CP)         │      (TASK-GOAL ?SUBTSK                             (b)
──────────────↓──────────────    │                 (ACHIEVE (?PRED ?DES ?VAL)))
(SEQ ?TO-BE-REVISED              │      (SUBTASK+ ?SUBTSK ?TSK)                        (c)
     (STABILIZE                  │      (HOLDS (TRACK ?DES ?OB) (DURING ?SUBTSK))      (d)
       ((λ (?OB) ?UNACH-GOAL)    │      (HOLDS ?UNACH-GOAL (END-TASK ?SUBTSK)))        (e)
        ?VAR)))                  │      (VALUE ?VAR (BEGIN-TASK ?SUBTSK) ?DES)         (f)
                                 └      (TASK-CODE-PATH ?SUBTSK ?CP)))                 (g)
```

Fig. 5.4. Example plan revision rule

```
((?SAFE-PROTECTEE-PLAN                          (AND (ORDERING-CONTEXT
     AT ?SAFE-PROTECTEE-CP)                              ?VIOLATOR ?TSK
 (?SAFE-VIOLATOR-PLAN                                    ?SAFE-VIOLATOR-CP
     AT ?SAFE-VIOLATOR-CP)                               ?SAFE-PROTECTEE-CP
 (?P-O-LOCUS-PLAN AT ?P-O-LOCUS-CP                       ?P-O-LOCUS-CP)
     == (PARTIAL-ORDER                               (EVAL (→NEWTAG 'PROTECTEE)
            ?STEPS !?ORDERINGS)))                          ?PROTECTEE-TAG)
─────────────────────↓─────────────────────          (EVAL (→NEWTAG 'VIOLATOR)
((:TAG ?PROTECTEE-TAG ?SAFE-PROTECTEE-PLAN)                ?VIOLATOR-TAG))
 (:TAG ?VIOLATOR-TAG ?SAFE-VIOLATOR-PLAN)
 (PARTIAL-ORDER ?STEPS
     (:ORDER ?VIOLATOR-TAG ?PROTECTEE-TAG)
     !?ORDERINGS))
```

Fig. 5.5. Plan revision rule that transforms more than one subplan.

TOR and ?PROTECTEE, and adding the constraint (:ORDER ?VIOLATOR-TAG ?PROTECTEE-TAG) to the smallest partial order including the protectee and violator.

5.4 XFRML — The Implementation

The predicates provided by XFRML are implemented as sets of retrieval procedures that extract information from linked data structures that constitute an execution scenario. These data structures are described in [127], although a few new ones have been added since that report. For example, because TASK-GOAL is such a ubiquitous predicate, XFRM's projector constructs a table during plan projection that maps goals to the tasks that are intended to achieve them. If this table did not exist the programmer would have to ask a query like

```
(AND (TASK ?TSK)
     (TASK-PLAN ?TSK (ACHIEVE ?GOAL-EXP))
     (VALUE ?GOAL-EXP (BEGIN-TASK ?TSK) G)),
```

to find a task for accomplishing a particular goal *G*. This query requires XFRML to enumerate all tasks in an execution scenario, extract the plan of the task, and evaluate the arguments of the plan in the context of the environment of the task. Instead of performing such an expensive search, the XFRML system can prove a relation (TASK-GOAL *TSK* *G*) with a fast associative retrieval from that table. Besides being very fast, queries of the form (TASK-GOAL ?TSK *G*) tend to have comparatively few solutions if *G* is at least partly instantiated. Therefore, having those predicates in the beginning of a query often speeds up the answering process tremendously.

The implementation of XFRML predicates that are procedural specialists is supported by the XFRML-predicate IS-BOUND. (IS-BOUND *EXP*) holds unless *EXP* is an uninstantiated XFRML variable. How XFRML predicates are implemented can be illustrated using an predicate (SUBTASK ?T1 ?T2) as an example. For its implementation we distinguish between different cases, depending on which of its arguments are instantiated. (SUBTASK ?T1 ?T2) would be defined through four XFRML rules: one for the case that ?T1 and ?T2 are unbound, one for the case that both are bound, the third one for the case that ?T1 is bound but not ?T2, and the last one for the other way around. A typical rule looks as follows:

```
(← (SUBTASK ?T1 ?T2)
   (AND (IS-BOUND ?T1)
        (IS-BOUND ?T2)
        (LISP-PRED SUBTASK? ?T1 ?T2))).
```

If both ?T1 and ?T2 are already bound, the clause determines the truth of (SUBTASK ?T1 ?T2) by applying the LISP-predicate to the bindings of ?T1 and ?T2.

The inference system performs different kinds of inferences such as Horn clause deductions and temporal inferences. Temporal inference is "lazy:" the temporal projector makes only the inferences necessary to decide which events occur in a projected execution scenario and does not infer the complete state of the world at each time point. If later, in the course of diagnosing a plan failure additional information is needed, the temporal projector will infer the necessary pieces of information on demand. Thus, in order to answer queries, the XFRML system might have to infer aspects of the state of the environment at a particular time point and do computationally expensive temporal inferences.

A planning cycle in XFRM can be performed quickly because the generation of execution scenarios by simulation-based projection requires little search. XFRM only has to search when the projector asks whether a particular state S holds at some time instant T. In this case the projector has to determine the states prior to T on which S conditionally depends. A second reason planning cycles are fast is that XFRML is implemented as a set of retrieval procedures that extract information from execution scenarios. The data structures of execution scenarios are linked to optimize typical inferences in failure diagnosis as, for instance, in the reconstruction of a variable value at a particular state of plan execution.

5.5 Discussion

A notation for plan revision, such as XFRML, is difficult to assess. In this section I will try to assess the significance of XFRML along several dimensions: the necessity of notations of this sort for robot control applications, the expressiveness of the language, the conciseness and coverage of the set of built-in predicates, the computational overhead caused by using the notation, and the performance of the delivery robot, which makes use of XFRML.

Autonomous robots acting in changing and partly unknown environments need a notation for plan revision methods because such robots exhibit behavior flaws for many different reasons and can robustify their behavior in many different ways.

Let me also forestall the objection that XFRML makes simple plan revisions complicated. Although the plan revision methods listed in section 5.3 are more complex than their counterparts used by other planners, they are also much more powerful. As I pointed out earlier, other planners assume the robot to be capable of referring to objects by their names or an one-to-one mapping between state transitions in the world and the the execution of plan steps. XFRML does not make these assumptions and therefore, has to infer the relationship between designators and the objects they designate and between plan pieces and the events they cause. The pay-off is that XFRML can infer that the robot has a faulty or ambiguous object description, that the object changed its appearance without the robot noticing, and so on.

XFRML can express all the common-sense plan revisions listed on page 101; we will formalize additional types of flaws and common-sense revision methods for avoiding them in chapter 6.3. Specifying these revision methods requires a plan revision language to represent not only the physical effects of plan execution, but also the process of plan interpretation, as well as temporal, causal, and teleological relationships between plan interpretation, the world, and the physical behavior of the robot. No other notation for reasoning about plans and their projections I know of covers all these aspects of plan execution.

We can also express planning operations performed by classical planning systems such as means-end analysis [135] or the operations of the SNLP planning algorithm can [119]. However, additional techniques are necessary to make a classical planning algorithm, such as SNLP [119], work in a reactive setting. In addition, I have implemented models for about twenty five types of behavior flaws and about forty transformation rules. Most of which cannot be expressed in other planning representations.

Another interesting project would be to see how XFRML would have to be extended to express the rules used by other transformational planners, such as those described in [156, 151]. HACKER's [156] plan revision methods can be expressed easily.

```
?CODE                              ┌ (AND (BUG PREREQUISITE-CLOBBERS-BROTHERS
                                   │             ?CODE ?T1 ?T2)
─────────────↓─────────────        │      (== ?CODE (PLAN ?STEPS ?CONSTRAINTS))
(PLAN                              │      (MEMBER (:ORDER ?T1 ?T2) ?CONSTRAINTS)
   ?STEPS                          │      (DELETE (:ORDER ?T1 ?T2) ?CONSTRAINTS
   ((:ORDER ?T2 ?T1)               │                ?RELAXED-CONSTRAINTS))
    !?RELAXED-CONSTRAINTS))        └
```

The transformation rule above is a reconstruction of a plan revision procedure in HACKER. The plan transformation is applicable if the plan ?CODE contains to steps ?T1 and in plan clobbers the precondition of another step ?T2 *(prerequisite clobbers brothers).* If the plan statement contains (:ORDER

?T1 ?T2) as an ordering constraint, then this ordering constraint is dropped and the ordering constraint (:ORDER ?T2 ?T1) is added instead.

GORDIUS' plan revision methods would look in XFRML as follows:

```
                                        ┌ (AND (CULPRIT-ASSUMPTION
                                        │          ?HYPOTHESIS
                                        │          (OCCURS ?A-TYPE ?ACTION))
                                        │      (== ?HYPOTHESIS
?HYPOTHESIS                             │          (PLAN ?STEPS ?CONSTRAINTS))
────────────────────↓──────────────────│      (PRECOND ?ACTION ?PRECOND)
 (PLAN                                  │      (OCCURS-IN ?STATE ?ACTION)
   ?NEW-STEPS                           │      (THNOT (HOLDS ?PRECOND ?STATE))
   ((:ORDER ?T2 ?T1)                    │      (DELETE (:TAG ?TAG ?ACTION) ?STEPS
    !?NEW-CONSTRAINTS))                 │              ?NEW-STEPS)
                                        │      (DELETE (:ORDER ?TAG ?*) ?CONSTRAINTS
                                        │              ?CONSTRAINTS1)
                                        │      (DELETE (:ORDER ?* ?TAG) ?CONSTRAINTS1
                                        └              ?NEW-CONSTRAINTS))
```

XFRML cannot handle such rules yet: the predicate CULPRIT-ASSUMPTION cannot be implemented based on the current data structures for execution scenarios. GORDIUS makes assumptions during plan generation and plan projection. In GORDIUS, projected plan failures are caused by invalid assumptions. GORDIUS' projector constructs a dependency structure that justifies the executability of plan steps and their effects. If a plan failure, a discrepancy between a desired and projected state, is detected, GORDIUS will propagate the problem backwards through the dependency structure to identify the invalid assumptions and regress goals to infer how to modify the assumptions.

To implement GORDIUS-like plan transformations, XFRM's projector would have to generate the same kinds of dependency structures. We have plans to extend XFRM this way. It is, however, an open question, how much of RPL could be handled by such dependency structures. Dependency structures depend on the projector being able to insert links from causes to their effects. This is comparatively easy as long as plans are partially ordered sets of discrete plan steps, as in GORDIUS' case. The identification of causes and their effects is much more difficult in RPL. In RPL, a specific behavior of the robot might be caused by subtle interactions of concurrent continuous control processes, or flawed behavior should be corrected by other processes. Such interactions and corrective behaviors are very difficult to represent as causal dependencies.

Another criterion for assessing XFRML is the conciseness of the notation. XFRML provides between fifty and sixty predicates; about half of them have been described in this chapter. Considering that the notation represents structured reactive plans, how the plans get interpreted, how the world changes, and the causal, teleological, and temporal relationships between these components, the number of built-in predicates is rather small. Many of the built-in predicates, especially the ones described in this chapter, are heavily used, that is, in most transformation rules we have written. XFRML is complete in the sense that any transformation that can be specified as a LISP program can also be specified as an XFRML transformation rule by using the EVAL predicate. However, as shown in the transformation rules listed in this and the next chapter these "escape" calls to LISP are exceptions and mainly used to perform small computations such as generating names for tags.

Besides expressiveness and conciseness, another criterion for the value of a notation is whether the overhead of using it is acceptable. In our application, the time for running an inference rule is negligible compared to the time for generating projections in the first place. For example, consider a routine plan for getting all objects from one place to another. The part of the default plan XFRM has to reason about is roughly 300 lines of RPL code long. The plan takes the system about four to five seconds to project. The projection contains more than eighty events that change the world state and more than five hundred tasks. Testing whether a projection satisfies the user's commands takes less than 0.03 seconds. Testing the applicability of the transformation rules from section 5.1 and installing the proposed revision takes much less than a second. There are two other speed-up techniques that the planning system does: caching temporal inferences (which are the most time-consuming ones) and organizing the applicability conditions of a set of transformation rules in a discrimination net.

In chapter 8 we conduct an experiment that suggests that XFRML is fast enough to improve the problem-solving behavior of a simple simulated robot while the robot performs its tasks.

Related Work

Related works in AI planning include works on representations for planning [112] and formalizations of plan interpretation [52], which reason either about plans and their (physical) effects or plan interpretation processes. Another line of planning research [28] focuses on meaning-preserving plan transformations, which assume perfect world models and unlimited computational resources. In addition, our approach contrasts with representations of engineered physical devices as considered in model-based troubleshooting [88]. XFRML reflects that reactive plans are not just connected components, or subplans, but contain control structures which coordinate subplans and recover from their failures. PRODIGY [44] and other partial-order planners use meta-languages for plans to implement search control rules. Khambhampati [102] describes the structure of plans explicitly to facilitate plan adaptation.

6. Forestalling Behavior Flaws

So far, we have described the tools for forestalling behavior flaws caused by structured reactive plans. In this chapter, we will explain how these tools are used. That is, we will enter into the particulars of the implementation of FAUST, a special-purpose planner in the XFRM framework (see section 3.3) that anticipates and forestalls flaws in the behavior of the robot. As we will see, FAUST can be specialized for particular robots, jobs, and environments by specifying types of behavior flaws and plan revision methods in XFRML.

The chapter is divided into three parts. The first part describes FAUST. The second part describes how FAUST makes the plan in the introductory example (section 1.3) more robust. Finally, the third part describes and discusses models and plan revision methods for several important categories of behavior flaws and describes how the models and revision methods are implemented in XFRML.

6.1 FAUST

FAUST, as a specialized planner in the XFRM framework (see chapter 3.4), consists of three components: (1) a specific class of bugs called *behavior-specification violations*, (2) a critic called the *behavior critic* that, given a projected execution scenario, finds behavior-specification violations for the top-level commands, and (3) an eliminator that, given a plan and a diagnosed behavior-specification violation, produces a set of candidate plans that might avoid the most severe flaws predicted for the original plan.

6.1.1 The Behavior Critic

The purpose of the behavior critic is to criticize the behavior the robot exhibits to accomplish its jobs. The critic gets a set of execution scenarios for a plan as its input and returns two results: the average score of the plan over the execution scenarios and a set of descriptions for the flaws that occurred in the projected scenarios together with the estimated severity of the flaws. In our discussion of the behavior critic we will detail

1. how behavior specifications for declarative statements are implemented and how FAUST's flaw-detection algorithm works;
2. how benefit models for declarative statements are specified and how the score of a plan based on an execution scenario is computed; and
3. how models of behavior flaws for declarative statements are defined and how FAUST's behavior flaw diagnosis algorithm works.

The Computational Structure of the Behavior Critic. The behavior critic performs the following computation steps (see figure 6.1). First, given the projected execution scenarios, the algorithm detects flaws in the projected behavior of the robot, represents these flaws as behavior-specification violations, and returns them as its result.

The next step marks the behavior flaws in the scenario for which the plan interpreter did not signal a failure as being undetected. In many cases, behavior flaws cannot be detected from the robot's sensors and therefore, the interpreter does not signal the plan failures. A *plan-generated* failure, also called cognizant plan failure [59], is signalled by the plan itself during execution or projection, by use of the RPL FAIL construct. Behavior flaws that have not been detected by the structured reactive controller have to be treated differently from those that have been detected. If they have not been detected the set up of additional monitoring processes is often required. The occurrence of detected failures, on the other hand, often requires the addition of control structures that specify how to recover from plan failures.

In the third step each behavior-specification violation is classified using the taxonomy of models of behavior flaws: for each behavior flaw the set of the most specific models and their instantiations are computed. In addition, the step merges different occurrences of the same flaw are merged into a single flaw description: FAUST represents two balls that are not at their destination for the same reason as the same flaw. Finally, the third step estimates the severity of behavior flaws, an estimation of how much the plan can be improved by eliminating the corresponding flaws in the plan.

The fourth step measures the average success of the plan in the given execution scenarios. It assigns a benefit to the plan for the number and the degree to which the user commands have been accomplished and subtracts a penalty that is a function of the time required to carry out the plan.

```
Behavior-Critic(exec-scenarios)
1  Flaws ← Detect-Flaws(exec-scenarios)
2  Mark-Undetected-Flaws(flaws)
3  Diagnose-Flaws(flaws)
4  Score ← Measure-Plan-Success(exec-scenarios)
5  return ⟨flaws, Score⟩
```

Fig. 6.1. Algorithm for detecting and diagnosing behavior flaws.

6.1.2 Detecting Behavior Flaws: Implementation

A behavior flaw is simply a violation of the constraints specified by the top-level commands. If a top-level command is not a declarative goal as, for instance, (GO ⟨*4,5*⟩), the top-level command is considered to be accomplished successfully if the execution of the subplan does not signal a plan failure. If the top-level command is a declarative goal, the condition of a successful accomplishment is stated in the goal's behavior specification. A behavior flaw is deduced by the planner by comparison of what was supposed to happen with what actually happened during execution or projection. Such an inference is possible only if the planner has models of what was supposed to happen, the behavior specifications, and what actually did. The second model, of what really "happened," is supplied by the projector. Deciding what happened from a projection is a matter of retrieval and inference.

Declarative goals can be viewed as logical sentences which are true if and only if the plan is executed successfully (this idea has been formalized by McDermott [122]). For example, the command *"get all the balls from location ⟨0,8⟩ to the location ⟨0,10⟩"* is just the sentence stating that all the blocks at location ⟨*0,8*⟩ now, will be at the location ⟨*0,10*⟩ before and until they are needed there; plus the tag: to be made true. Thus, in order to check whether a plan has been projected to be successful, FAUST only has to check whether the projection satisfies the logical sentence representing the command.

The idea of behavior specifications being logical statements that are true if and only if the corresponding top-level command is accomplished successfully has a drawback when applied in the context of planning. Detecting that a plan has flaws is not enough to guide the search for better plans effectively. The number of transformations that can be applied to a structured reactive plan is huge. One heuristic to deal with the huge search space for better plans is to exploit the information why the plan failed to guide the search for better plans [155]. Applying this idea, the flaw detection step does not merely check whether a plan is predicted to be successful but predicts what could go wrong and summarizes the detected flaws as bugs. Through this modification, search becomes more effective because the planner generates only those plan hypotheses intended to eliminate the expected flaws.

Rather than formalizing the concept of intended behavior, we formalize the concept of "behavior flaw." That is, we specify that a ball that has been at ⟨*0,8*⟩ when the command was given, but is not at ⟨*0,10*⟩ when the robot is done has a flaw of the behavior for the command *"get all the balls from location ⟨0,8⟩ to the location ⟨0,10⟩"*. In this view, a behavior specification is satisfied if no behavior flaw occurs.

Formalizing Behavior Flaws in XFRML. The concept "behavior flaws" is represented by the predicate (BEH-SPEC-VIOLATIONS *TSK BUGS*), which is true if *BUGS* describes all violations of the behavior specification of *TSK* in the loaded execution scenario. The plan writer defines XFRML Horn clause rules for this predicate as part of the RPL construct descriptions of declarative statements. Horn clause rules for specifying behavior-specification violations make use of three other predicates: (ABSTRACT-TASK-PLAN *TSK EXP*), (REPRESENTS-FLAW *BUG FLAW TSK*), and (GOAL-MEANING *ROBOT-GOAL WORLD-GOAL*).

(ABSTRACT-TASK-PLAN *TSK EXP*) holds if *EXP* is the most abstract RPL expression of the plan for *TSK*. For example, if the subplan *PL* for performing *TSK* has the form

```
(REDUCE (REDUCE (ACHIEVE g)
                 ...
                 DEFAULT-PLAN)
        (SEQ (REDUCE (ACHIEVE g)
                     ...
                     DEFAULT-PLAN)
             (STABILIZE g))
        PLAN-TRANSFORMATION),
```

then the query (ABSTRACT-TASK-PLAN *TSK* ?EXP) binds (ACHIEVE *g*) to ?EXP. Thus, the predicate ABSTRACT-TASK-PLAN can be used to extract the declarative goals from subplans that have been expanded and revised by the planner.

Another predicate we make heavy use of is (REPRESENTS-FLAW *BUG F T*), which holds if *BUG* describes the flaw *F* in the execution of task *T*. *F* is a XFRML statement, for example, (THNOT (HOLDS *g* (TASK-END TOP))). (REPRESENTS-FLAW *BUG F T*) can be defined by the following XFRML Horn clause rules:

```
(DEF-FACT-GROUP REPRESENTS-FLAW
    (← (REPRESENTS-FLAW ?BUG ?FLAW ?TSK)
       (THNOT (IS-BOUND ?BUG))
       (IS-BOUND ?FLAW ?TSK)
       (EXECUTION-SCENARIO ?ES)
       (EVAL (→BEH-SPEC-VIOLATION ?FLAW ?TSK ?ES)
             ?BUG))
    (← (REPRESENTS-FLAW ?BUG ?FLAW ?TSK)
       (IS-BOUND ?BUG)
       (EVAL (SLOT-VALUE FLAW ?BUG) ?FLAW)
       (EVAL (SLOT-VALUE TASK ?BUG) ?TSK))        )
```

The first rule specifies that if ?FLAW and ?TSK are bound, but not ?BUG, then create a data structure that describes the behavior-specification violation ?FLAW of task ?TSK and bind the data structure to the variable ?BUG. If, on the other hand, ?BUG is bound then ?FLAW and ?TSK have to unify with the values of the slots FLAW and TASK of the behavior-specification violation bound to ?BUG (see second rule).

The last predicate (GOAL-MEANING *ROBOT-GOAL WORLD-GOAL*) means that *ROBOT-GOAL* is a goal in the language used by the structured reactive controller that corresponds to a goal state *WORLD-GOAL* in an execution scenario. The difference is that goals in the structured reactive controller are expressed using designators, while goals in execution scenarios use the objects themselves. Thus if

```
(GOAL-MEANING (LOC DESIG-23 ⟨0,8⟩) (LOC BALL-23 ⟨0,8⟩))
```

holds the subplan (ACHIEVE (LOC DESIG-23 ⟨0,8⟩)) will accomplish its goal if it causes (LOC BALL-23 ⟨0,8⟩) to become true.

Behavior Flaws of ACHIEVE-Statements. Now that we have the necessary predicates we can explain the XFRML Horn clause rules that define behavior flaws of declarative goals.

```
(DEF-FACT-GROUP BEH-SPEC-FOR-ACHIEVE
    (← (BEH-SPEC-VIOLATIONS ?TSK (?BUG))
        (ABSTRACT-TASK-PLAN ?TSK (ACHIEVE ?GOAL-EXP))
        (VALUE ?GOAL-EXP (BEGIN-TASK ?TSK) ?ROBOT-GOAL)
        (GOAL-MEANING ?ROBOT-GOAL ?WORLD-GOAL)
        (THNOT (HOLDS ?WORLD-GOAL (END-TASK ?TSK)))
        (REPRESENTS-FLAW
            ?BUG (THNOT (HOLDS ?WORLD-GOAL (END-TASK ?TSK))) ?TSK))
    (← (BEH-SPEC-VIOLATIONS ?TSK (?BUG))
        (ABSTRACT-TASK-PLAN
            ?TSK (ACHIEVE ?GOAL-EXP :UNTIL (?RELATIVE ?CONSUMER)))
        (VALUE (?GOAL-EXP ?OB) ?ROBOT-GOAL)
        (GOAL-MEANING ?ROBOT-GOAL ?WORLD-GOAL)
        (THNOT (HOLDS ?WORLD-GOAL (?RELATIVE ?CONSUMER)))
        (REPRESENTS-FLAW
            ?BUG
            (THNOT (HOLDS ?WORLD-GOAL (?RELATIVE ?CONSUMER)))
            ?TSK))
```

Basically, the first rule says: If a task has the purpose to achieve some goal, which does not hold when the plan is completed, then there is a bug that the goal does not hold. However, we cannot state the rule that simple because the planner has to start with a task, which is an object in the plan interpretation and compute the goal, a desired state in the world. Thus, the rule has to infer the purpose of the plan by finding the most abstract declarative statement, which the plan reduces. Since declarative statements evaluate their arguments, the planner has to reconstruct the particular binding of the arguments upon starting the task. Finally, it has to substitute the designators in the declarative statement with the objects they were produced from. Therefore, the rule has to be stated as: if

1. the most abstract plan for ?TSK has the form (ACHIEVE ?GOAL-EXP);
2. the value of ?GOAL-EXP in the program environment of ?TSK is ?ROBOT-GOAL;
3. the meaning of ?ROBOT-GOAL in the execution scenario is ?WORLD-GOAL;

4. ?WORLD-GOAL does not hold at the end of ?TSK; and
5. ?BUG represents the flaw (THNOT (HOLDS ?WORLD-GOAL (END-TASK ?TSK))) of task ?TSK

then the set of behavior-specification violations of ?TSK is the set containing ?BUG as its only element. The second rule is similar; the only difference being that the abstract task plan for ?TSK is an achievement goal with a keyword parameter :UNTIL.

Behavior Flaws of ACHIEVE-FOR-ALL-Statements. The next XFRML rule computes the behavior-specification violations of ACHIEVE-FOR-ALL-goals. The first three conditions compute the description ?DESC of the objects for which the goal ?GOAL is to be achieved. The fourth condition is true if ?OBS is the set of objects in the projected execution scenario that satisfy ?DESC in the beginning of task ?TSK, but not ?GOAL at the end of ?TSK. The fifth condition generates for each of the objects ?OBS a behavior flaw description with the flaw (THNOT (HOLDS ?UNACHIEVED (END-TASK ?TSK))) and collects these flaw descriptions in ?BUGS.

```
(← (BEH-SPEC-VIOLATIONS ?TSK ?BUGS)
   (ABSTRACT-TASK-PLAN
        ?TSK (ACHIEVE-FOR-ALL ?DESC-EXP ?GOAL-EXP))
   (VALUE ?DESC-EXP (BEGIN-TASK ?TSK) (λ (?ARG) ?DESC))
   (VALUE ?GOAL-EXP (BEGIN-TASK ?TSK) (λ (?ARG) ?GOAL))
   (SETOF ?ARG
          (AND (HOLDS ?DESC (BEGIN-TASK ?TSK))
               (THNOT (HOLDS ?GOAL (END-TASK ?TSK))))
          ?OBS)
   (SETOF ?BUG
          (AND (MEMBER ?OB ?OBS)
               (SUBSTITUTION ((?OB ?ARG)) ?GOAL ?UNACHIEVED)
               (REPRESENTS-FLAW
                  ?BUG
                  (THNOT (HOLDS ?UNACHIEVED (END-TASK ?TSK)))
                  ?TSK))
          ?BUGS))
```

Behavior Flaws of PERCEIVE-Statements. Parts of the definition of behavior flaws of the PERCEIVE-statement are listed below. There are two kinds of behavior-specification violations: first, the subplan returns FALSE while there are objects in the robot's view that satisfy *desig*, that is, the robot overlooks objects; second, the robot perceives an object as matching *desig* that does not in reality match *desig*, that is, sensor noise corrupts the perceptions of the robot.

```
(← (BEH-SPEC-VIOLATIONS ?TSK ?BUGS)
   (ABSTRACT-TASK-PLAN ?TSK (PERCEIVE ?DESIG))
   (OVERLOOKED-FAILURES ?TSK ?O-BUGS)
   (MISPERCEIVED-FAILURES ?TSK ?M-BUGS)
   (UNION ?BUGS ?O-BUGS ?M-BUGS))
```

The rule for determining the overlooked objects is applicable if, in a hypothetical execution scenario, the PERCEIVE-subplan returns FALSE while there are objects in view that satisfy *desig*. For each of the overlooked objects a behavior flaw description of the form (THNOT (AND (RETURNED-VALUE ?TSK ?A-DESIG) (CORRESPONDS ?A-DESIG ?OVERLOOKED-OB))) is generated that says the subplan has not returned a designator that corresponds to a particular overlooked object.

```
(← (OVERLOOKED-FAILURES ?TSK ?BUGS)
    (ABSTRACT-TASK-PLAN ?TSK (PERCEIVE ?DESIG))
    (RETURNED-VALUE ?TSK FALSE)
    (PERCEPTUAL-DESCRIPTION ?DESIG (λ (?X) ?DESCR))
    (SETOF ?X (HOLDS ?DESCR (DURING ?TSK)) ?OVERLOOKED)
    (SETOF ?BUG
           (AND (MEMBER ?OVERLOOKED-OB ?OVERLOOKED)
                (REPRESENTS-FLAW
                    ?BUG
                    (THNOT (AND (RETURNED-VALUE ?TSK ?A-DESIG)
                                (CORRESPONDS
                                    ?A-DESIG ?OVERLOOKED-OB)))
                    ?TSK))
           ?BUGS))
```

The rule for handling misperceived objects is applicable if, in a hypothetical execution scenario, the PERCEIVE-subplan returns a designator that corresponds to an object that does not match the designator ?DESIG.

```
(← (MISPERCEIVED-FAILURES ?TSK (?BUG))
    (ABSTRACT-TASK-PLAN ?TSK (PERCEIVE ?DESIG))
    (PERCEPTUAL-DESCRIPTION ?DESIG (λ (?X) ?DESCR))
    (RETURNED-VALUE ?TSK ?A-DESIG)
    (CORRESPONDS ?A-DESIG ?AN-OB)
    (SUBSTITUTION ?AN-OB ?X ?DESCR ?PERCEIVED-DESCR)
    (THNOT (HOLDS ?PERCEIVED-DESCR (DURING ?TSK)))
    (REPRESENTS-FLAW
        ?BUG (AND (RETURNED-VALUE ?TSK ?A-DESIG)
                  (CORRESPONDS ?A-DESIG ?AN-OB)
                  (THNOT (HOLDS ?PERCEIVED-DESCR (DURING ?TSK))))
        ?TSK))
```

The Algorithm for Behavior Flaw Detection. Having defined the predicate BEH-SPEC-VIOLATIONS the planner can now use the query (?- (BEH-SPEC-VIOLATIONS *TLC-TSK* ?BUGS) to obtain all behavior-specification violations of a top-level command *TLC-TSK* in a given execution scenario. The flaws in the behavior of the robot in a given execution scenario *ES* is computed as follows:

```
(WITH-LOADED-EXECUTION-SCENARIO ES
    (UNION (?-VAR ?BUGS
                  (AND (TLC-TASK ?TLC ?TSK)
                       (BEH-SPEC-VIOLATIONS ?TSK ?BUGS))))).
```

When the query succeeds, *BUGS* is bound to a set of flaw descriptions that name a top-level task and describe the observed flaw. A plan succeeds if none of the behavior specifications of its top-level tasks are violated.

6.1.3 Diagnosing the Causes of Behavior Flaws: Implementation

Behavior flaw descriptions produced by the detection step specify what went wrong, as for instance, (THNOT (HOLDS *g* (END-TASK *t*))) or (AND (RETURNED-VALUE *t-1 d-25*) (CORRESPONDS *d-25 o-30*) (THNOT (HOLDS (COLOR *o-30* RED) (DURING *t-1*)))), but do not explain what caused the flaw. Providing this explanation is the purpose of the diagnosis step that takes behavior flaw descriptions and produces diagnoses of the causes of the behavior flaws (cf. [151]). For the purpose of diagnosing behavior flaws, categories of behavior flaws are described by models that specify characteristic conditions and specializations of the flaw categories. A diagnosis of the causes of behavior flaws consists of a reference to the subplan that should be revised, a behavior flaw model, and the instantiation of the model that explains the flaw.

FAUST uses a classification algorithm that takes a behavior flaw description and computes the set of most specific flaw models the flaw matches. The CLASSIFY-BUG algorithm, shown below, is a straightforward recursive algorithm.

```
Classify-Bug(fm, bdg)
1 Bdgs ← Test-Characteristic-Cond(fm, bdg)
2 Specs ← Specializations(fm)
3 Diags ← {}
4 for each fm' in Specs
5     do for each bdg' in Bdgs
6         do Diags ← Append(Diags, Classify-Bug(fm', bdg')
7 if ¬empty?(Diags)
8    then return Diags
9    else return fm × Bdgs
```

Models of Behavior Flaws. FAUST's diagnosis algorithm uses a taxonomy of behavior flaw models. A category of behavior flaws is described by a model that consists of a characteristic condition, a method for the determination of the critical subplan, the specializations of the model, and a set of plan revision rules.

Figure 6.2 shows the model of ACHIEVE-FOR-ALL flaws. The model is a specialization of the model *declarative goal flaw* and has *clobbered subgoal* and *unachieved subgoal* as its specializations. The CHARACTERISTIC-CONDITION slot specifies the conditions a behavior flaw has to meet in order to be an ACHIEVE-FOR-ALL *flaw*. In our case, the condition is that the plan of the task that violates its behavior specification has the form (ACHIEVE-FOR-ALL

behavior flaw model*ACHIEVE-FOR-ALL FLAW*	
Generalization	Declarative Goal Flaw
Specializations	Clobbered Subgoal, Unachieved Subgoal
Characteristic Condition	(AND (REPRESENTS-FLAW ?BUG ?FLAW ?TSK) (ABSTRACT-RPL-EXP ?TSK (ACHIEVE-FOR-ALL ?DESCR-EXP ?GOAL-EXP)) (== ?FLAW (THNOT (HOLDS ?UNACHIEVED-GOAL (END-TASK ?TSK-2)))) (VALUE ?DESCR-EXP (BEGIN-TASK ?TSK) ?DESCR) (== ?DESCR (λ (?D-ARG) ?D)) (VALUE ?GOAL-EXP (BEGIN-TASK ?TSK) ?GOAL) (== ?GOAL (λ (?G-ARG) ?G)))
To Be Revised	(?-VAR ?CP (TASK-CODE-PATH ?TSK ?CP))
Revisions	{ AFA-FLAW.REV-1 AFA-FLAW.REV-2 }
Refinements	{ }

Fig. 6.2. Behavior flaw model for "ACHIEVE-FOR-ALL flaws."

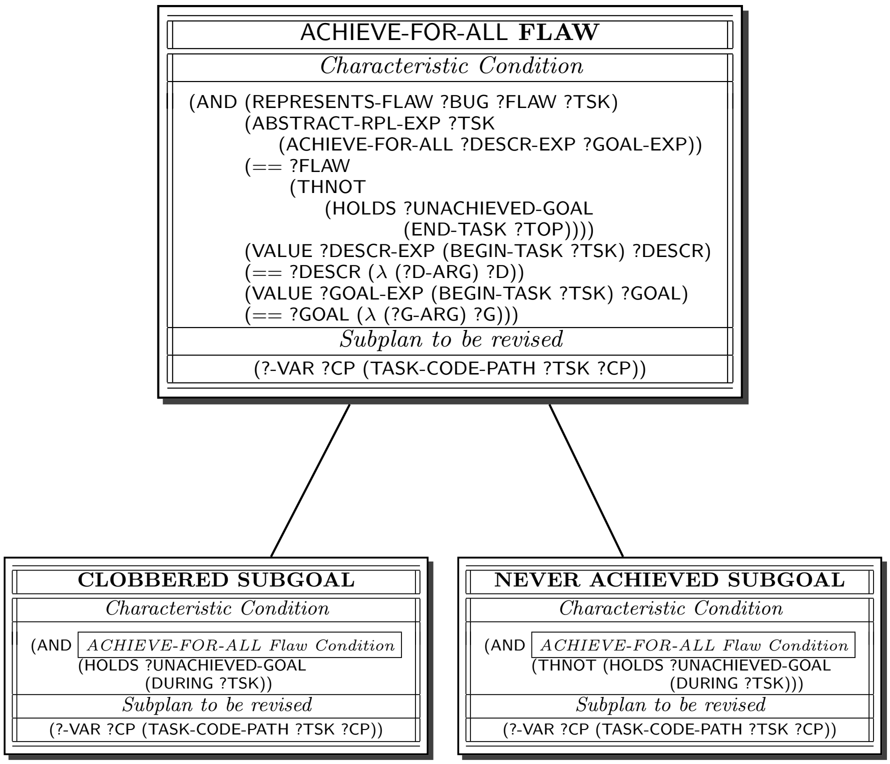
?DESCR-EXP ?GOAL-EXP) and the description of the flaw has the form (THNOT (HOLDS ?UNACHIEVED-GOAL (END-TASK ?TSK-2))); the purpose of the remaining conditions is the instantiation of pattern variables that will be needed later in the diagnosis process. Thus the diagnosis process can determine whether a detected behavior flaw with the description *bug-25* is an ACHIEVE-FOR-ALL *flaw* by

binding *bug-25* to the variable ?BUG and using the characteristic condition as a XFRML query:

```
(WITH-LOADED-EXECUTION-SCENARIO ES
   (?- (AND (REPRESENTS-FLAW ?BUG ?FLAW ?TSK)
            (ABSTRACT-RPL-EXP
               ?TSK (ACHIEVE-FOR-ALL ?DESCR-EXP ?GOAL-EXP))
            (== ?FLAW (THNOT (HOLDS ?UNACHIEVED-GOAL
                                    (END-TASK ?TSK-2))))
            (VALUE ?DESCR-EXP (BEGIN-TASK ?TSK) ?DESCR)
            (== ?DESCR (λ (?D-ARG) ?D))
            (VALUE ?GOAL-EXP (BEGIN-TASK ?TSK) ?GOAL)
            (== ?GOAL (λ (?G-ARG) ?G)))
       :BINDING ((?BUG bug-25)))).
```

The slot TO-BE-REVISED specifies a XFRML query that computes (the code path of) the smallest subplan in which the cause of the flaw could be localized. Storing this subplan as part of the diagnosis result serves two purposes: (1) it specifies a subplan that has to be revised as a default and (2) it is used for checking whether two occurrences of behavior flaws are of the same flaw.

To explain the role of the REFINEMENTS-slot suppose that a behavior flaw in a plan *P* is of the category FAILED-SUBPLAN. In this case, FAUST does not further specialize the model but identifies and diagnoses the flaws in the subplan. To accomplish this, the characteristic condition of FAILED-SUBPLAN



```
(AND (REPRESENTS-FLAW ?BUG ?FLAW ?TSK)
     (ABSTRACT-RPL-EXP ?TSK
        (ACHIEVE-FOR-ALL ?DESCR-EXP ?GOAL-EXP))
     (== ?FLAW
         (THNOT
            (HOLDS ?UNACHIEVED-GOAL
                   (END-TASK ?TOP))))
     (VALUE ?DESCR-EXP (BEGIN-TASK ?TSK) ?DESCR)
     (== ?DESCR (λ (?D-ARG) ?D))
     (VALUE ?GOAL-EXP (BEGIN-TASK ?TSK) ?GOAL)
     (== ?GOAL (λ (?G-ARG) ?G)))

(?-VAR ?CP (TASK-CODE-PATH ?TSK ?CP))
```

```
(AND ACHIEVE-FOR-ALL Flaw Condition
     (HOLDS ?UNACHIEVED-GOAL
            (DURING ?TSK))

(?-VAR ?CP (TASK-CODE-PATH ?TSK ?CP))
```

```
(AND ACHIEVE-FOR-ALL Flaw Condition
     (THNOT (HOLDS ?UNACHIEVED-GOAL
                   (DURING ?TSK)))

(?-VAR ?CP (TASK-CODE-PATH ?TSK ?CP))
```

Fig. 6.3. Specializations of the model "ACHIEVE-FOR-ALL flaw."

contains a condition of the form (BEH-SPEC-VIOLATIONS *st ?ST-BUGS*) and the REFINEMENTS-slot contains the variable ?ST-BUGS. In order to specialize flaws of the type FAILED-SUBPLAN, FAUST's classification algorithm calls itself recursively with *?ST-BUGS* as its argument, and returns the results of the recursive calls as the diagnosis of the original flaw.

The REVISIONS-slot of the behavior flaw model specifies the names of plan transformation rules that can be applied to the plan that produced the execution scenario *es* in order to eliminate the detected ACHIEVE-FOR-ALL flaw. An example of such a revision rule is one that modifies the plan such that the subplan for the ACHIEVE-FOR-ALL goal is executed twice. This revision rule would improve a plan if, for instance, in a specific situation the ACHIEVE-FOR-ALL plan has a fifty percent chance of overlooking the objects for which the goal has to be achieved. Another transformation rule could change the overall plan such that the top-level command will be completed after all the other jobs are already finished. This revision would help if the behavior flaws are caused by interferences among subplans.

The Taxonomy of Behavior Flaws. The plan transformation rules sketched above are only helpful for a small fraction of ACHIEVE-FOR-ALL flaws. In order to cover a broader range of these flaws, the diagnosis algorithm has to obtain more information about why a certain ACHIEVE-FOR-ALL flaw occurred. For this purpose the planner has a taxonomy of models of ACHIEVE-FOR-ALL flaws which we will describe below.

The model ACHIEVE-FOR-ALL flaw has two specializations (see figure 6.3): clobbered subgoal and never achieved subgoal. The characteristic condition of clobbered subgoal is the conjunction of the characteristic condition of ACHIEVE-FOR-ALL flaw and (HOLDS ?UNACHIEVED-GOAL (DURING ?TSK)). That is, the subgoal that does not hold when the plan is completed must hold for some time during the execution of the subplan for the ACHIEVE-FOR-ALL

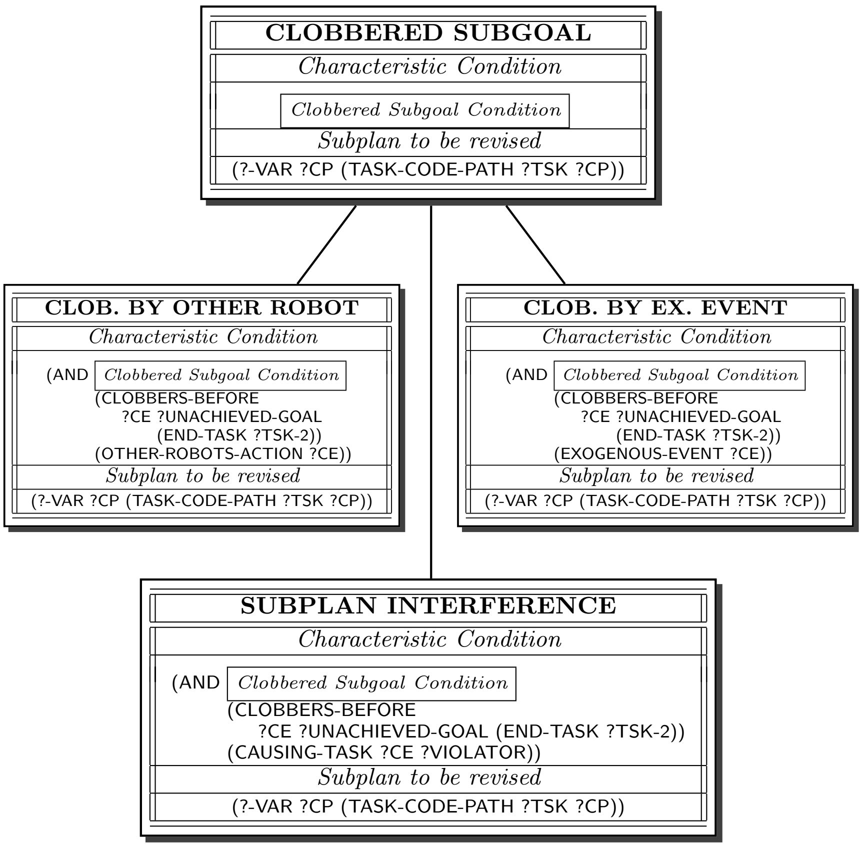


Fig. 6.4. Specializations of the model "clobbered subgoal."

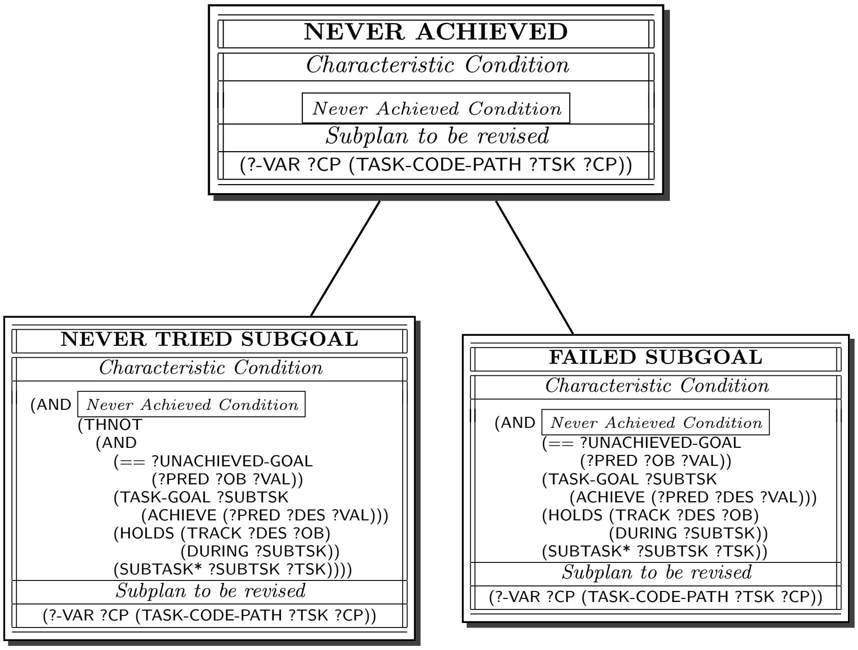


Fig. 6.5. Specializations of the model "never achieved subgoal."

goal. The characteristic condition of never achieved subgoal is the conjunction of the characteristic condition of ACHIEVE-FOR-ALL flaw and (THNOT (HOLDS ?UNACHIEVED-GOAL (DURING ?TSK))).

Figure 6.4 shows the specializations of the model "clobbered subgoal." The characteristic conditions of all three specializations determine the last event before the end of ?TSK-2 that causes the subgoal not to hold any longer and differ only with respect to the rest of the condition: whether the event was caused by another robot ((OTHER-ROBOTS-ACTION ?CE)), by an exogenous event ((EXOGENOUS-EVENT ?CE)), or the execution of another subplan ((CAUSING-TASK ?CE ?VIOLATOR)). Similar taxonomies of behavior flaws can be specified for maintenance, perception and other declarative tasks.

Measuring the Success of a Plan. An autonomous robot should try to accomplish as many of its jobs in as little time as possible. This kind of behavior is achieved by incorporating a utility model into the planner that assigns benefits to the robot for accomplishing jobs from which costs proportional to the time spent for doing so are subtracted. The utility model defines the score of a plan in a given execution scenario *es* with respect to a set of top-level commands as follows:

$$\text{score}(es) = \left(\sum_{tlc \in TLCS} \text{benefit}(tlc, es) \right) - (\text{end}(es) \text{ - } \text{start}(es)) \times \text{time-cost}.$$

The amount of benefits the robot gets for accomplishing individual commands depends on the kinds of commands and the degree to which they are accomplished. In general, methods for measuring the benefit a robot obtains from accomplishing a job g work according to the following pattern. First, they determine what is to be achieved and when. Second, they compute the intrinsic value of accomplishing the job g [67]. For instance, the intrinsic value of a delivery could depend on the weight and size of the object to be delivered and the distance from the current location of the object to its destination. Third, the benefit estimators measure the extent to which a top-level command is achieved at different time instants give partial credit for parts of jobs that are accomplished after a deadline is passed. Benefit models that are specific to declarative statements combine the results of the first three steps into a number that summarizes the benefit the robot gains from accomplishing g. The benefits a robot obtains in a particular execution scenario is the sum of the benefits over all top-level command [81, 82]. Hanks and Haddawy [82] describe a utility model for a robot that includes concepts like temporally scoped goals, partial goal satisfaction, and the cost of resource consumption. Many of their ideas are incorporated here.

The programmer has to provide benefit models for the declarative statements. Benefit models are specified using the BENEFIT predicate. Like behavior specifications and behavior flaw models, benefit models of declarative statements are implemented as XFRML Horn clause rules.

```
(DEF-FACT-GROUP BENEFIT-FOR-ACHIEVE

    (← (BENEFIT ?TLC ?BENEFIT)
       (TLC-EXPRESSION ?TLC (ACHIEVE ?GOAL))
       (HOLDS ?GOAL (END-TASK TOP))
       (GOAL-VALUE ?GOAL ?BENEFIT))

    (← (BENEFIT ?TLC 0)
       (TLC-EXPRESSION ?TLC (ACHIEVE ?GOAL))
       (THNOT (HOLDS ?GOAL (END-TASK TOP))))
```

These two rules specify the benefit the robot obtains from accomplishing a top-level command of the form (ACHIEVE g). If the goal state g holds when the overall plan is completed, the benefit is the value of goal g; if it does not hold, the benefit is zero.

For the delivery robot in the DELIVERYWORLD, we have defined the benefit obtained from accomplishing a command of the form (ACHIEVE-FOR-ALL d g) is proportional to the number of goal instances the robot has accomplished and the value of the individual goals.

```
(← (BENEFIT ?TLC ?BENEFIT)
   (TLC-EXPRESSION ?TLC (ACHIEVE-FOR-ALL (λ (?X) ?D) (λ (?X) ?G)))
   (SETOF ?X (AND (HOLDS ?D (BEGIN-TASK TOP))
                  (HOLDS ?G (END-TASK TOP)))
          ?OBJS-WITH-ACHIEVED-GOALS)
   (EVAL (LENGTH ?OBJS-WITH-ACHIEVED-GOALS)
         ?NR-ACHIEVED-SUBGOALS)
   (GOAL-VALUE ?D ?G ?G-VAL)
   (EVAL (* ?NR-ACHIEVED-SUBGOALS ?G-VAL) ?BENEFIT))
```

The utility of a plan in a given execution scenario is computed as follows:

$$\sum_{\text{tlc} \in \text{TLCS}} \text{(?-VAR ?BENEFIT (TLC-BENEFIT tlc ?BEN))}$$
$$\text{— RESOURCE-COST(ES).}$$

6.1.4 The Bug Class "Behavior-Specification Violation"

Failures or flaws in the behavior of the robot are described using the data structure BEH-SPEC-VIOLATION, which is shown in figure 6.6. Two methods are associated with BEH-SPEC-VIOLATION: one for determining whether two bugs are the same and the other one for estimating the severity of a bug.

Determining the Equality of Behavior Flaws. As we said before, FAUST represents two balls that are not at their destination for the same reason as the same behavior flaw. For the purpose of transformational planning two flaws are the same if they can be eliminated by the same revisions. Merging instances of the same flaws cuts down the search space. There is another advantage to summarizing the instances of the same flaw: avoiding a behavior flaw that occurs several times is often more important than removing one that occurs only once. This bias in the search for better plans is implemented by summing up the severity of the flaw instances.

FAUST considers two behavior flaws to be the same if they are instances of the same models and are thought to be caused by the same subplan. Thus, if two goals g_1 and g_2 of a top-level task have not been achieved because a sensor program *sp* has overlooked the objects for which the goals should be achieved, the two occurrences of the flaws count as multiple occurrences of the same flaw. If, however, one object was overlooked and the other one

FLAW	A logical statement representing a difference between a behavior in the scenario and a desired behavior.	
SEVERITY	An estimate of how much the plan can be improved by eliminating the flaw.	
TASK	The task in which the flaw occurred.	
IS-RECOGNIZED	Whether the flaw was recognized by the robot controller.	
DIAGNOSIS	FLAW-MODEL	Name of the behavior flaw model that the flaw is an instance of.
	CODE-PATH	Subplan that caused the flaw.
	BDGS	Bindings for the logical variables in the characteristic condition.
	SCENARIO	Execution scenario.

Fig. 6.6. Data structure BEH-SPEC-VIOLATION.

disappeared before it was sensed, the two flaws are not considered to be the same. They are also not considered to be the same kind of flaw if they occur in different subplans. A sufficient criterion is that the same kind of flaw occurs in the same subplan.

Determining the Severity of Behavior Flaws. The severity of a behavior flaw is a guess of how much the plan could be improved by eliminating the flaw of the plan represented by the flaw description. We take the value of the unachieved goal as an estimate of the severity. This typically overestimates the severity because most plans resulting from plan revisions are slower. However, by making behavior flaws look more severe than they are we can make sure that the planner tries to eliminate them [57].

There is another complication: transforming a plan to avoid behavior flaws, in particular by introducing additional ordering constraints, might introduce new behavior flaws. Approaches to deal with this complications are described in [148, 127].

6.1.5 The Elimination of Behavior Flaws

The algorithm for producing new candidate plans is straightforward. It takes a plan and a diagnosed behavior flaw as its argument and returns a set of new plans.

```
GENERATE-CANDIDATE-PLANS(PLAN, FLAW)
1 for each D in DIAGNOSES(FLAW)
2    do for each BDG in BDGS(D)
3       do for each TR in REVISION-RULES(BEHAVIOR-FLAW-MODEL(D))
4          do collect APPLY-REVISION(TR, PL, BDG)
```

6.2 The Plan Revisions for the Example

After having detailed the implementation of the special-purpose planner FAUST, we will now see how the delivery robot with the assistance of FAUST has accomplished the top-level commands in the example described in section 1.3. Figure 6.7 shows the basic structure of the structured reactive controller that we have used in this example, which is largely fixed. Given the two top-level commands

```
(ACHIEVE-FOR-ALL (λ (X) (AND (CAT X BALL) (LOC X ⟨5,5⟩)))
                 (λ (X) (LOC X ⟨2,5⟩)))
```

and

```
(ACHIEVE-FOR-ALL (λ (X) (AND (CAT X BALL) (LOC X ⟨2,5⟩)))
                 (λ (X) (LOC X ⟨2,7⟩))),
```

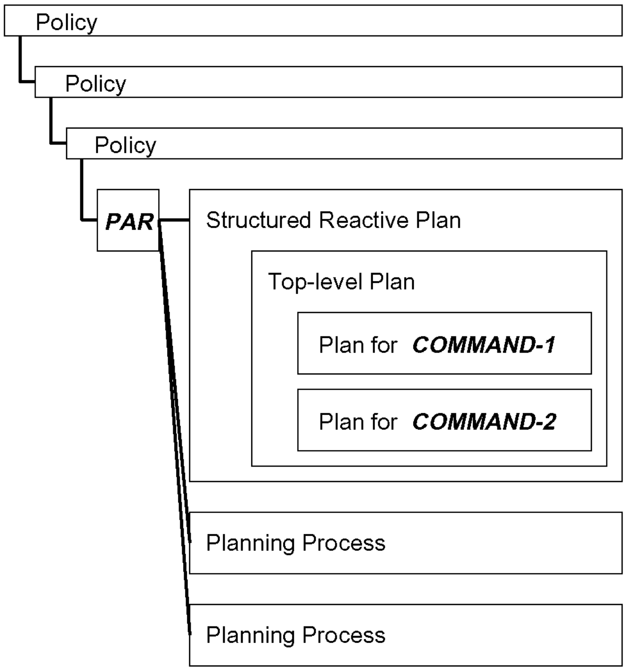


Fig. 6.7. Configuration of the structured reactive controller used for solving this problem.

the initial plan generator has retrieved routine plans for the commands from the plan library, expanded the declarative goals in the routine plans recursively, and substituted the resulting plan fragments for the top-level commands.

The RPL code for the structured reactive controller is shown in figure 6.8. The subplans marked with a "*" (the PERCEIVE and ACHIEVE subplans) are hidden. The length of the RPL plan that the planner has reasoned about is about three hundred lines of RPL code. For now, we will ignore the pieces of the structured reactive controller that control planning processes and the synchronization between planning and execution. We will discuss the implementation of these parts in the next chapter.

An execution scenario for the routine plan (figure 6.8) is shown in figure 6.9. The figure shows parts of the projected task network and timeline. The timeline consists of a sequence of time instants. With each time instant there is associated a set of events and a set of occasions. The events represent what happened at this time instant. The occasions describe the state of the world at this instant. The timeline shows that in the beginning there are two balls at location $\langle 5,5 \rangle$ and one ball at $\langle 2,5 \rangle$. The behavior specifications implied by the top-level commands tell FAUST the two balls at $\langle 5,5 \rangle$ should be at $\langle 2,5 \rangle$ and the one at $\langle 2,5 \rangle$ at $\langle 2,7 \rangle$. However, in the execution scenario, all three

```
(:TAG AGENT-CONTROLLER
      (WITH-POLICY (CHECK-SIGNPOSTS-WHEN-NECESSARY)
        (WITH-POLICY (CHECK-OBS-REMOVED-FROM-BOXES)
          (WITH-POLICY (SCAN-FOR-OTHER-ROBOTS)
            (PAR (:TAG RC
                       (PLAN
                         ((:TAG MAIN
                                (TOP-LEVEL
                                  (:TAG COMMAND-1
                                        (REDUCE (ACHIEVE-FOR-ALL (λ (AND (CAT X BALL)
                                                                         (LOC X ⟨5,5⟩)))
                                                                 (λ (X) (LOC X ⟨2,5⟩)))
                                        (LOOP
                                             (PROCESS A-DELIVERY
                                                 (VALVE-REQUEST A-DELIVERY WHEELS)
                                                 (IF (NULL CURR-GOAL-VAR-1)
                                                     (!= CURR-GOAL-VAR-1
                                                     *(PERCEIVE
                                                          (λ (X) (AND (LOC X ⟨5,5⟩)
                                                                      (CAT X BALL))))))
                                                 (IF (IS-DESIG CURR-GOAL-VAR-1))
                                                     *(ACHIEVE ((λ (X) (LOC X ⟨2,5⟩))
                                                     CURR-GOAL-VAR-1)))
                                         UNTIL (NULL CURR-GOAL-VAR-1)
                                             (!= CURR-GOAL-VAR-1 ())))))
                                  (:TAG COMMAND-2
                                        (REDUCE (ACHIEVE-FOR-ALL (λ (AND (CAT X BALL)
                                                                         (LOC X ⟨2,5⟩)))
                                                                 (λ (X) (LOC X ⟨2,7⟩)))
                                        (LOOP
                                             (PROCESS A-DELIVERY
                                                 (VALVE-REQUEST A-DELIVERY WHEELS)
                                                 (IF (NULL CURR-GOAL-VAR-1)
                                                     (!= CURR-GOAL-VAR-1
                                                     *(PERCEIVE
                                                          (λ (X) (AND (LOC X ⟨2,5⟩)
                                                                      (CAT X BALL))))))
                                                 (IF (IS-DESIG CURR-GOAL-VAR-1))
                                                     *(ACHIEVE ((λ (X) (LOC X ⟨2,7⟩))
                                                     CURR-GOAL-VAR-1)))
                                         UNTIL (NULL CURR-GOAL-VAR-2)
                                             (!= CURR-GOAL-VAR-2 ())))))
      (TRY-ALL (:TAG ELIMINATE-SUBPLAN-INTERFERENCES
                      (RUNTIME-PLAN RC DONE?* BETTER-PLAN?* BEST-PLAN-SO-FAR*)))
                      (WAIT-FOR (OR NEW-ROBOT* OUTDATED-WORLD-MODEL)))
      (TRY-ALL (:TAG PLANNER (ANTICIPATE-AND-FORESTALL-FAILURES RC))
               (:TAG SWAPPER (EXECUTE-BEST-REACTIVE-CONTROLLER RC))
               (WAIT-FOR (END-TASK RC))))))))
```

Fig. 6.8. RPL code for the structured reactive controller.

balls end up at ⟨2,7⟩. Thus the behavior flaw detection step returns two plan behavior flaw descriptions: (THNOT (HOLDS (LOC BALL-1 ⟨2,5⟩) (END-TASK TOP*))) and (THNOT (HOLDS (LOC BALL-2 ⟨2,5⟩) (END-TASK TOP*))). The sketchy execution scenario in figure 6.9 details what happens to the ball BALL-1: at time instant TIMEINSTANT-28 the state (LOC BALL-1 ⟨2,5⟩) becomes true and persists until TIMEINSTANT-37. Later (LOC BALL-1 ⟨2,7⟩) becomes true and persists till the end of the execution scenario.

Thus, when FAUST retrieves the characteristic condition of task interference flaw

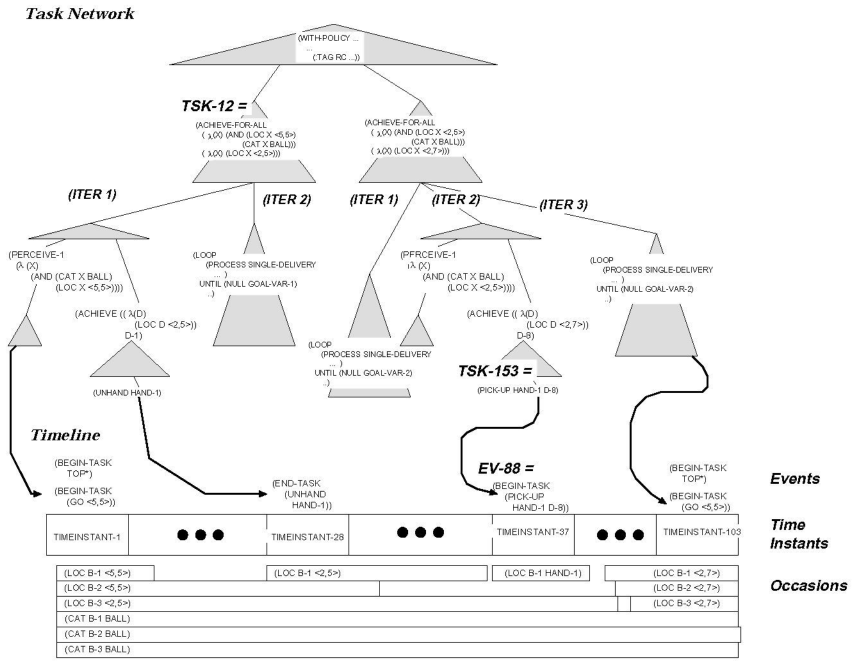

Task Network
(WITH-POLICY ... ... (TAG RC ...))
TSK-12 =
(ACHIEVE-FOR-ALL (λ(X) (AND (LOC X <5,5>) (CAT X BALL))) (λ(X) (LOC X <2,5>)))
(ACHIEVE-FOR-ALL (λ(X) (AND (LOC X <2,5>) (CAT X BALL))) (λ(X) (LOC X <2,7>)))
(ITER 1)
(ITER 2)
(ITER 1)
(ITER 2)
(ITER 3)
(PERCEIVE-1 (λ (X) (AND (CAT X BALL) (LOC X <5,5>))))
(LOOP (PROCESS SINGLE-DELIVERY ...) UNTIL (NULL GOAL-VAR-1) ..)
(PERCEIVE-1 (λ (X) (AND (CAT X BALL) (LOC X <2,5>))))
(LOOP (PROCESS SINGLE-DELIVERY ...) UNTIL (NULL GOAL-VAR-2) ..)
(ACHIEVE ((λ(D) (LOC D <2,5>)) D-1)
(ACHIEVE ((λ(D) (LOC D <2,7>)) D-8)
(LOOP (PROCESS SINGLE-DELIVERY ...) UNTIL (NULL GOAL-VAR-2) ..)
TSK-153 =
(PICK-UP HAND-1 D-8)
(UNHAND HAND-1)
Timeline
(BEGIN-TASK TOP*)
(BEGIN-TASK (GO <5,5>))
(END-TASK (UNHAND HAND-1))
EV-88 =
(BEGIN-TASK (PICK-UP HAND-1 D-8))
(BEGIN-TASK TOP*)
(BEGIN-TASK (GO <5,5>))
Events
TIMEINSTANT-1
TIMEINSTANT-28
TIMEINSTANT-37
TIMEINSTANT-103
Time Instants
(LOC B-1 <5,5>)
(LOC B-1 <2,5>)
(LOC B-1 HAND-1)
(LOC B-1 <2,7>)
(LOC B-2 <5,5>)
(LOC B-2 <2,7>)
(LOC B-3 <2,5>)
(LOC B-3 <2,7>)
(CAT B-1 BALL)
(CAT B-2 BALL)
(CAT B-3 BALL)
Occasions

Fig. 6.9. An example execution scenario for the RPL plan in figure 6.8.

```
(AND (REPRESENTS-FLAW ?BUG ?FLAW ?TSK)
     (ABSTRACT-RPL-EXP ?TSK
        (ACHIEVE-FOR-ALL ?DESCR-EXP ?GOAL-EXP))
     (== ?FLAW (THNOT (HOLDS ?UNACHIEVED-GOAL
                             (END-TASK ?TOP))))
     (VALUE ?DESCR-EXP (BEGIN-TASK ?TSK) ?DESCR)
     (== ?DESCR (λ (?D-ARG) ?D))
     (VALUE ?GOAL-EXP (BEGIN-TASK ?TSK) ?GOAL)
     (== ?GOAL (λ (?G-ARG) ?G)))
     (HOLDS ?UNACHIEVED-GOAL (DURING ?TSK))
     (CLOBBERS-BEFORE ?CE ?UNACHIEVED-GOAL (END-TASK ?TSK-2))
     (CAUSING-TASK ?CE ?VIOLATOR)
```

and evaluates it as an XFRML query on the execution scenario, the query will succeed with a binding that contains EV-88 for ?CE, TSK-12 for ?TSK, and TSK-153 for ?VIOLATOR. Since the behavior flaw model of task interference does not have any specializations, the model and the bindings for the variables in the characteristic condition are returned as the result of the diagnosis step. One of the transformation rules stored under the flaw model "task interference" is the rule ORDER-INTERFERING-SUBPLANS, which is shown in figure 5.5. The bug eliminator applies this transformation rule with the initial binding returned by the diagnosis step. Checking the application condition successfully results in binding SAFE-PROTECTEE-PATH to the code path of the plan executed for task TSK-12, SAFE-VIOLATOR-PATH to the code path of

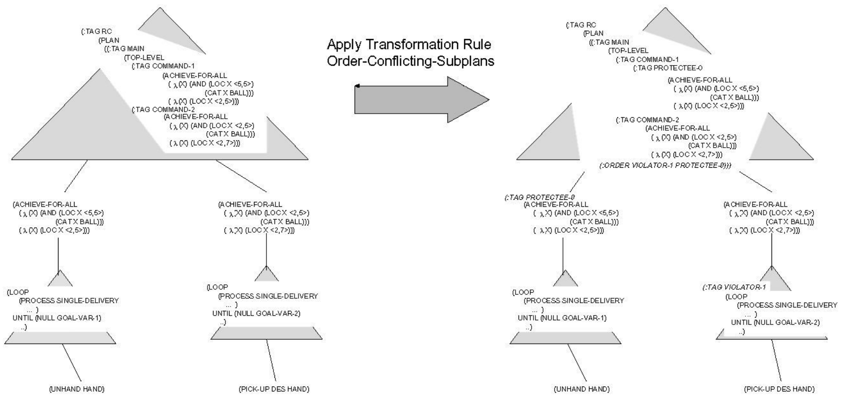


Fig. 6.10. Modification of the code tree by the plan revision rule.

the LOOP subplan under command COMMAND-2, and ?PO-LOCUS-CP to the subplan that constitutes the main partial order. The revision rule then assigns a name to the ?SAFE-PROTECTEE-PLAN and the SAFE-VIOLATOR-PLAN and adds the ordering constraint (:ORDER ?VIOLATOR-TAG PROTECTEE-TAG) to the main partial order. The corresponding modifications of the code tree are shown in figure 6.10.

The corresponding modification in the default plan are shown below.

```
(:TAG RC
      (PERFORM (PARTIAL-ORDER ((:TAG MAIN ...)))
       BY (PARTIAL-ORDER
            ((:TAG MAIN
                  (TOP-LEVEL
                   (:TAG COMMAND-1
                         (:TAG PROTECTEE-0
                               (REDUCE (ACHIEVE-FOR-ALL
                                             '(λ (X) (AND (LOC X ⟨8,8⟩)
                                                          (CAT X BALL)))
                                             '(λ (X) (LOC X ⟨0,10⟩))) ...)))
                   (:TAG COMMAND-2
                         (REDUCE (ACHIEVE-FOR-ALL
                                       '(λ (X) (AND (LOC X ⟨0,10⟩)
                                                    (CATEGORY X BALL)))
                                       '(λ (X) (LOC X ⟨0,12⟩)))
                                     (:TAG VIOLATOR-1
                                           (LOOP ... (PICK-UP ***) ...))))))
            (:ORDER VIOLATOR-1 PROTECTEE-0)))
       BECAUSE TASK-INTERFERENCE.REV-1))
```

Based on projected execution scenarios for the revised plan, the latter seems to be better than the original plan. One of the planning processes then substitutes the improved plan for the task RC for the original one. We will describe how this is done in the next chapter.

```
(:TAG RC
      (PERFORM (PARTIAL-ORDER ((:TAG MAIN ...)))
       BY (PARTIAL-ORDER
            ((:TAG MAIN
                   (TOP-LEVEL
                    (:TAG COMMAND-1
                          (:TAG PROTECTEE-0
                                (REDUCE ... ...)))
                    (:TAG COMMAND-2
                          (REDUCE
                            (ACHIEVE-FOR-ALL
                             '(λ (X) (AND (LOC X (- - -))
                                          (CATEGORY X BALL)))
                             '(λ (X) (LOC X ⟨0,12⟩)))
                            (:TAG VIOLATOR-1
                                  (LOOP
                                    γ
                                    (PERFORM *(ACHIEVE
                                                 ((λ (X) (LOC X ⟨0,12⟩)) OB)
                                     BY (SEQ *(ACHIEVE ((λ (X) ...) OB))
                                             *(STABILIZE
                                                 ((λ (X) (LOC X ⟨0,12⟩)) OB)
                                     BECAUSE CLOBBERED-BY-ROBOT)
                                    ...))))))
            (:ORDER VIOLATOR-1 PROTECTEE-0)))
       BECAUSE TASK-INTERFERENCE.REV-1))
```

When executing the revised plan the robot perceives the cleaning robot. Projecting its revised plan in the presence of the cleaning robot results in an execution scenario in which BALL-3 does not end up at location ⟨2,7⟩ because the cleaning robot has put it away. As a revision, the planner proposes to stabilize the goal state (LOC BALL-3 ⟨2,7⟩) by locking up the ball immediately after achieving the state. The plan resulting from this revision is shown below.

6.3 Some Behavior Flaws and Their Revisions

Many autonomous robots use special purpose planning capabilities such as motion planning [111], navigation planning, or revising cleaning plans to direct the robot into areas with high expected concentration of trash [74].

FAUST can be tailored to particular robots, jobs, and environments by defining behavior flaw models and plan revision methods for avoiding the flaws. For example, because the delivery robot in the DeliveryWorld should perform its deliveries in time and deliver objects successfully even when their descriptions are ambiguous, we have equipped the structured reactive controller of the delivery robot with means to forestall deadline violations and problems caused by ambiguous descriptions. We have defined two additional flaw models: “missed deadline” and “perceptual confusion” and plan revision methods for avoiding these flaws. In this section we will describe how such models and plan revision methods are formalized in XFRML.

6.3.1 Perceptual Confusion

A perceptual confusion is a flaw that causes the robot to achieve a goal for an object that is not the intended one. Perceptual confusion flaws are caused by ambiguous description of the intended objects.

Perceptual Confusion

Characteristic Condition

```
(AND (THNOT (HOLDS ?UNACHIEVED-GOAL (DURING ?TSK)))
     (THNOT (HOLDS (PHYSICAL-TRACK ?DES ?OB) (DURING ?SUBTSK)))
     (TASK-GOAL ?PERC-TSK (PERCEIVE ?DES))
     (THNOT (HOLDS (VISUAL-TRACK ?A-DESIG ?OB) (END-TASK ?PERC-TSK)))
     (HOLDS (VISUAL-TRACK ?DISTRACTOR-DESIG ?DISTRACTOR)
            (END-TASK ?PERC-TSK))
     (SETOF ?PERCEIVED-OB
            (HOLDS (LOCAL-DESIG ?A-DESIG ?PERCEIVED-OB)
                   (END-TASK ?PERC-TSK))
            ?PERCEIVED-OBS)
     (MEMBER ?DISTRACTOR ?PERCEIVED-OBS)
     (MEMBER ?OB ?PERCEIVED-OBS))
```

Subplan to be revised

```
(?-VAR ?CP (TASK-CODE-PATH ?TSK ?CP))
```

A behavior flaw is caused by a perceptual confusion if

- at the end of a perception task (PERCEIVE ?DES) the robot tracks an object ?DISTRACTOR instead of the intended object ?OB (conditions 3 – 5) and
- both ?OB and ?DISTRACTOR were recognized by the perception subplan (conditions 6 – 8).

One way to deal with perceptual confusion in the context of an achievement task is to achieve the goal for each object satisfying an ambiguous object description. The advantage of this revision is that the goal in question is certainly achieved; the disadvantage is that the robot might waste more resources than necessary. Another problem is that the more objects the robot manipulates, the more likely it is that interferences between subplans occur.

```
(?TO-BE-REVISED AT ?CP)                  ┌
                                         │ (AND (EVAL (DESIG→DESCR ?DES)
───────────────↓───────────────          │              ?DESIG-DESCRIPTION)
(ACHIEVE-FOR-ALL                         │      (TASK-CODE-PATH ?TSK ?CP)))
    ?DESIG-DESCRIPTION                   │
    (λ (?OB) ?UNACHIEVED-GOAL))          └
```

If the designator was non-ambiguous in the initial state of the execution scenario and became ambiguous due to some projected changes in the world, the robot can mark the object as its very first action and use the mark in order to distinguish the object from distracting objects:

```
((?TO-BE-REVISED AT ?CP)
 (?MPO == (PARTIAL-ORDER
                ?STEPS !?ORDERINGS)
       AT ?MPO-CP))
──────────────────↓──────────────────
((ACHIEVE-FOR-ALL
    (λ (?D-ARG) (AND (LOC ?D-ARG ⟨?X,?Y⟩))
                     (MARK ?D-ARG ?MARK)))
    ?GOAL)
 (PARTIAL-ORDER
    ((:TAG ?MARK-OBJECTS
          (ACHIEVE-FOR-ALL
             (λ (?D-ARG)
                (AND (MARK ?D-ARG NONE)
                     !?CONJUNCTS))
             (λ (?D-ARG)
                (MARK ?D-ARG ?MARK))))
     !?STEPS)
    (:ORDER ?MARK-OBJECTS MAIN)
    !?ORDERINGS))

(AND (AT-LOCATION ?VAR ?D ?X ?Y)
     (== ?D (AND !?CONJUNCTS))
     (TASK-CODE-PATH ?TSK ?CP)
     (MAIN-PARTIAL-ORDER ?MPO-CP)
     (EVAL (->NEWTAG
               'MARK-CHANGING-OBJS)
           ?MARK-OBJECTS)
     (EVAL (->NEWSYMBOL 'MARK)
           ?MARK))
```

If another subplan has changed the world in a way that made the designator ambiguous (for instance, by putting similar objects next to the object to be manipulated), then the subplan that causes the confusion could be executed after the goal for the target object is achieved:

```
((?SAFE-PERCEIVER-PLAN
     AT ?SAFE-PERCEIVER-CP)
 (?SAFE-CONFUSER-PLAN
     AT ?SAFE-CONFUSER-CP)
 (?P-O-SUPER-PLAN
     AT ?P-O-SUPER-CP
     == (PARTIAL-ORDER
            ?STEPS !?ORDERINGS)))
──────────────↓──────────────
((:TAG ?PERCEIVER-TAG
       ?SAFE-PERCEIVER-PLAN)
 (:TAG ?CONFUSER-TAG
       ?SAFE-CONFUSER-PLAN)
 (PARTIAL-ORDER
    ?STEPS
    (:ORDER ?PERCEIVER-TAG
            ?CONFUSER-TAG)
    !?ORDERINGS))

(AND (DESIG-PROPS-AT
         ?DES
         (BEGIN-TASK ?PERC-TSK)
         ?PROPS)
     (EVAL (DESIG→PL-DESCR ?DES 'VAR)
           ?DESIG-DESCR)
     (HOLDS ?DESIG-DESCR
            (BEGIN-TASK ?PERC-TSK))
     (START-TIME
         ?DESIG-DESCR ?START
         (BEGIN-TASK ?PERC-TSK))
     (THNOT (== ?START 0))
     (CAUSING-TASK ?START ?CONFUSER)
     (ORDERING-CONTEXT
         ?PERC-TSK ?CONFUSER
         ?SAFE-PERCEIVER-CP
         ?SAFE-CONFUSER-CP
         ?P-O-SUPER-CP)
     (EVAL (→NEWTAG 'PERCEIVER)
           ?PERCEIVER-TAG)
     (EVAL (→NEWTAG 'CONFUSER)
           ?CONFUSER-TAG))
```

If the ambiguity occurred because the description of the object was too sparse, the robot could examine the object in the beginning to obtain a more detailed description that would distinguish it from similar ones.

6.3.2 Missed Deadlines

To handle tight deadlines competently, we have equipped the delivery robot with a behavior flaw model "missed deadline." A "missed deadline" flaw occurs if the robot has a top-level command with a deadline ?DL and the task for accomplishing the top-level command finishes after ?DL has passed (see below).

Missed Deadline
Characteristic Condition
(AND (TLC-TASK ?TLC ?TLC-TSK) (TLC-DEADLINE ?TLC ?DL) (TASK-END ?TLC-TSK ?TLC-TSK-END) (DATE ?TLC-TSK-END ?DONE) (> ?DONE ?DL))
Subplan to be revised
(?-VAR ?CP (TASK-CODE-PATH ?TLC-TSK ?CP))

The following plan transformation rules use a user defined predicate (BLOWN-DEADLINE ?TLC ?TLC-TSK ?DONE ?DL ?ES ?BUG), which when ?BUG is bound to a data structure of the bug class missed deadline the predicate binds the slot values of the data structure to the variables ?TLC, ?TLC-TSK, ?DONE, ?DL, and ?ES.

The first plan transformation rule for missed deadlines proposes that if a subplan for accomplishing a top-level command misses its deadline, the robot should try to execute the subplan as early as possible.

```
                       ┌ (AND (BLOWN-DEADLINE
                       │           ?TLC ?TLC-TSK ?DONE ?DL ?ES ?BUG)
    (?PL AT ?CP)       │      (MAIN-PARTIAL-ORDER
                       │           (PARTIAL-ORDER ?STEPS !?ORDERINGS)
                       │           ?CP)
 ─────────↓─────────   │      (TLC-NAME ?TLC ?TLC-NAME)
                       │      (SETOF ?ORDER-CONSTRAINT
 (PARTIAL-ORDER        │             (AND (TLC-TASK ?OTHER-TLC ?OTHER-TLC-TSK)
    ?STEPS             │                  (COULD-PRECEDE ?OTHER-TLC-TSK ?TSK)
    !?ORDERINGS        │                  (TLC-NAME ?OTHER-TLC ?OTHER-TLC-NAME)
    !?CONSTRAINTS)     │                  (== ?ORDER-CONSTRAINT
                       │                      (:ORDER ?TLC-NAME ?OTHER-TLC-NAME)))
                       └             ?CONSTRAINTS))
```

Another revision rule proposes not to waste time and give up accomplishing the command if the deadline cannot be met.

```
 (?PL AT ?CP)  ┌
               │ (AND (BLOWN-DEADLINE
 ─────↓─────   │           ?TLC ?TLC-TSK ?DONE ?DL ?ES ?BUG)
               │      (TASK-CODE-PATH ?TLC-TSK ?CP))
 (NO-OP)       └
```

A third revision is to signal a plan failure of the class "missed deadline" if a plan for a top-level command misses its deadline to give the planner an opportunity to replan or inform the supervisor.

```
 (?PL AT ?CP)                                ┌
                                             │ (AND (BLOWN-DEADLINE
 ───────────────────↓───────────────────     │           ?TLC ?TLC-TSK ?DONE
 (WIITH-POLICY                               │           ?DL ?ES ?BUG)
      (SEQ (WAIT-UNTIL-DATE ?DL)             │      (TASK-CODE-PATH ?TLC-TSK ?CP))
           (FAIL :CLASS MISSED-DEADLINE))    │
    ?PL)                                     └
```

Yet another change in the course of action is to ask the robot to do as much as it can until the deadline is passed to obtain benefits for partly accomplishing the top-level command. This transformation is particularly useful for ACHIEVE-FOR-ALL commands.

```
(?PL AT ?CP)                 ┌
                             │ (AND (BLOWN-DEADLINE
─────────────↓───────────────│        ?TLC ?TLC-TSK ?DONE ?DL ?ES ?BUG)
(TRY-ALL                     │      (TLC-DEADLINE ?TLC ?DL)
   (WAIT-UNTIL-DATE ?DL)     │      (TASK-CODE-PATH ?TLC-TSK ?CP))
   ?PL)                      └
```

Or, the planner could suggest that the robot does as much as it can now and then does the rest later when it has spare time.

```
((?PL AT ?CP)                     ┌
 (?MPO                            │
 == (PARTIAL-ORDER                │
        ?STEPS !?ORDERINGS)       │
 AT ?MPO-CP))                     │ (AND (BLOWN-DEADLINE
────────────────↓─────────────────│        ?TLC ?TLC-TSK ?DONE ?DL ?ES ?BUG)
((TRY-ALL                         │      (TASK-CODE-PATH ?TLC-TSK ?CP)
    (WAIT-UNTIL-DATE ?DL)         │      (MAIN-PARTIAL-ORDER ?MPO-EXP ?MPO-CP)
    ?CP)                          │      (EVAL (-¿NEWTAG 'COMPLETE-TLC)
 (PARTIAL-ORDER                   │            ?COMPLETE-TLC))
    ((:TAG ?COMPLETE-TLC ?CP)     │
     !?STEPS))                    │
    (:ORDER MAIN ?COMPLETE-TLC)   │
    !?ORDERINGS))                 └
```

Other possible revision methods include the use of faster (a possibly more risky) plans (e.g, higher robot speed); the computation of better schedules for the plan; or the avoidance of interruptions of the plan that missed the deadline.

Which transformations should be applied depends on the utility model used by the robot and the particular situation.

6.4 Summary and Discussion

In this chapter we used XFRML for the implementation of FAUST, a planner in the XFRM framework that anticipates and forestalls behavior flaws. The top-level of FAUST's behavior critic, the flaw classification algorithm, and algorithm for the application of plan transformation rules are written in LISP (about three pages of code); all the rest is written in XFRML. In the past, these planning procedures have been completely written in LISP causing the resulting codes to be difficult to implement and debug. I have implemented the procedures in XFRML, which enabled me to express them in terms of abstract relations on RPL plans, plan interpretations, and the world and relations between plan interpretation and the world. XFRML enabled me to develop models of the behavior flaws and plan transformation rules in several months. This is very fast considering the complexity of the revisions and the concurrent, reactive plans they were revising, and that the transformations were tested for the delivery robot in the DeliveryWorld.

This chapter has also detailed the implementation of the taxonomy of behavior flaws that is used for forestalling flaws in the behavior of the delivery robot in the DeliveryWorld. This taxonomy of behavior flaws is still far away from being a theory for revising the plans of autonomous robots. However, it makes some important contributions: the taxonomy is complete, that is any behavior flaw of the delivery robot is described by at least one behavior flaw model in the taxonomy and the taxonomy contains only models of general categories of behavior flaws. Many of these behavior flaws cannot be handled by other planning systems.

Unfortunately, completeness and generality of the taxonomy do not guarantee its usefulness, in particular if the robot has only scarce computational resources. The current version the planner can only perform a small number of projections of candidate plans (more than fifteen in most cases) in the time the robot needs to execute the plan. Therefore, it is important that behavior flaw models do not contain a large number of plan revision methods that are applicable in the same situations and that plan revision methods have a good chance of proposing better candidate plans when they are applicable. I expect that the projection can be made considerably faster so that these considerations will be less of a factor.

I believe that in the long run we will develop more general and elegant theories of forestalling flaws in the behavior of autonomous robots. Having languages that are capable of characterizing a wide range of realistic behavior flaws and classifications of behavior flaws are important steps towards developing such theories. Another important step towards such theories is the careful analysis of the relationship between assumptions made about the robot, its jobs, and its environment and the flaws in the behavior of the robot. Giving up assumptions means for a planner that it has to deal with additional behavior flaws. A planner that does not make the assumption that the robot can refer to objects by their names will need a more detailed model of how the robot perceives objects and uses its perceptions to manipulate objects. If this model describes perception as imperfect and incomplete, the planner might have to consider that a plan might fail because the robot overlooks objects or cannot correctly perceive an object's visual properties. Also, if the robot perceives only what it needs to perform its tasks or the visual properties of objects are not unique then plans might fail because of ambiguous object descriptions, and so on. Such a theory has to account for the fact that as we drop assumptions about the robot, its jobs, and its environment, more and more kinds of behavior flaws become possible.

7. Planning Ongoing Activities

The delivery robot in the DELIVERYWORLD performs its jobs in a largely unknown and changing environment and acquires important and previously missing information during plan execution. Acting competently in such circumstances requires a flexible interaction between planning and execution: a robot perceiving an empty box should think about whether that box could help it to perform its current jobs more efficiently. If the box is helpful, the robot then will need to plan how to use it. Or, for instance, if the robot sees another robot, it should reason about whether that robot might interfere with some of its plans and revise these plans if necessary. Or, if a robot is to deliver objects from one location to another, it should postpone planning the errand until it knows how many objects it is to deliver. In all these examples the robot has to respond to risks and opportunities by (re)planning subtasks and postponing planning as long as information is missing. The robot also has to stop planning if the information used for planning is detected to be outdated, or install plan revisions without causing problems for the robot's ongoing activities.

Accomplishing its jobs reliably and efficiently under such circumstances requires the delivery robot to plan in a flexible and focused manner: it has to reflect on how to perform its jobs while accomplishing them, focus on critical aspects of important subtasks, and ignore irrelevant aspects of their context. In addition, it has to postpone planning when it lacks information or reconsider its course of action when noticing opportunities, risks, or execution failures. Finally, the robot must integrate revisions for its plans smoothly into its ongoing activities.

I call this kind of planning that flexibly interacts with plan execution *local planning of ongoing activities.* Besides its flexibility, local planning of ongoing activities is also computationally attractive: it allows an embedded robot planning system to plan in realtime by extracting and solving small planning problems on the fly and integrating their solutions smoothly into the robot's overall plan.

The problem of how to coordinate planning and execution is central to the effective use of planning to control autonomous robots. It has been studied for a long time, but still remains an open research issue. I investigate the problem within a particular framework for planning and action: the XFRM framework [127, 21], which is based on the following principles:

- *Action:* the course of action is specified by *structured reactive plans.*
  - Structured reactive plans specify how the robot is to respond to sensory input in order to accomplish its jobs.
  - Structured reactive plans are written in a language (RPL), which provides constructs for sequencing, conditionals, loops, local variables, and subroutines. The language also provides high-level concepts (interrupts, monitors) that can be used to synchronize parallel actions, to make plans reactive, etc.
- *Planning:* XFRM applies a planning technique called *transformational planning of reactive behavior.*
  - Routine plans for the robot's jobs are stored in a plan library.
  - Planning is the revision of structured reactive plans. Its goal is to avoid flaws in the problem-solving behavior of the robot that become predictable as the robot learns new information about its world.
  - XFRM solves the following planning problem: given a structured reactive plan and a partial description of the world, transform the plan into another one with higher expected utility.

The XFRM framework and RPL allow for a very flexible and tight integration of planning and execution, which is based on three ideas:

1. Extending RPL with three constructs: one for starting planning processes on subplans, one for installing revisions in a concurrently executed plan, and one for postponing planning of individual subplans.
2. Designing the constructs such that they constitute an interface between RPL and planning processes that is identical to the interface between RPL and continuous control processes like moving or grasping (see chapter 2.2.2).
3. Implementing planning processes such that they behave like sensing routines that monitor aspects of the world: whenever something interesting happens during planning, such as a better plan has been found or the search for better plans is completed, the planning process sends a signal. In addition, planning processes provide and continually update their best results in program variables.

Simultaneously running computational processes that plan and control the physical behavior of an autonomous robot raise some difficult technical problems — most notably, the problems of

- turning a detected contingency, the data structures that describe the robot's observations, and the structured reactive plan into a planning problem such that (1) solutions to the planning problem cause a more robust behavior of the robot in the contingent situation and (2) the planning problem can be solved fast; and
- integrating the revision proposed by a planning effort into a partly executed activity.

We will see that structured reactive plans can be designed such that simple algorithms can solve these difficult problems.

Due to the uniformity of the interfaces between RPL/planning processes and RPL/physical control processes and the control structures provided by RPL, a plan writer can concisely specify a wide spectrum of interactions between planning and execution in a behavior-based robot control framework. RPL becomes a single high-level language that can handle both planning and execution actions. Providing the extensions for local planning of ongoing activities as primitives in RPL enables RPL's control structures to control not only the robot's physical actions but also its planning activities. It also allows RPL plans to synchronize threads of plan execution with local planning processes and specify the communication between them.

This chapter is organized in two parts. The first part describes the constructs added to RPL in order to allow for local planning of ongoing activities and discusses their implementation. The second part describes how we can realize important patterns of interaction between planning and execution and implement some robot control architectures in extended RPL. We conclude by assessing these results and discussing related work.

7.1 Extending RPL

Before we discuss the RPL extensions, let us recall some of the RPL concepts that we need for the realization of local planning of ongoing problem-solving behavior.

The first one is the concept of *concurrency*. Various control structures in RPL create, start, and terminate RPL processes that can run concurrently. RPL also provides *fluents*, program variables that can signal changes of their values. Using fluents a programmer can write pieces of RPL plans that react to asynchronous events. For instance, the statement (WHENEVER *(= F V) PL*) is a loop that executes a plan *PL* every time the fluent *F* gets the value *V*. The statement (WAIT-FOR *(= F V)*) waits until the fluent *F* gets the value *V*. Fluents can be set by sensing processes, physical control routines, and assignment statements. In the context of runtime planning RPL control structures give the robot controller the ability to create and control local planning processes and specify the synchronization and communication between planning and execution processes using fluents.

*Task*s are another important concept in the implementation of local planning. A *task* is a piece of a plan *PL* the system intends to execute. The interpretation of complex tasks creates subtasks. For example, executing the plan $P =$ (N-TIMES 5 (GRASP *OB*)) generates a task *T* with plan (N-TIMES 5 (GRASP *OB*)) and five subtasks *(iter I T)* for $I = 1,\ldots,5$ with plan (GRASP *OB*). Within the RPL interpreter this notion of task corresponds to a data structure — similar to a stack frame in standard programming languages. This data structure stores the computational state of interpreting *PL*. In RPL plans, tasks are first-class objects and can therefore be passed as arguments of planning and plan swapping processes. The statement (:TAG A (GO 4 5)) statement creates a variable A and sets the value of A to the task "going to location *⟨4,5⟩*". Tasks can also be indexed relative to other tasks. The expression (SUBTASK (ITER *3*) *T*) returns the task for the third iteration of *T* in the example above. Tasks are created on demand, i.e., when they are executed, inspected, referred to, or revised. Some tasks are calls to physical control routines. They start continuous feedback loops that control the robot's effectors and continue immediately with the interpretation of the RPL plan. Control routines use fluents for storing their execution status and the results of their execution and update these fluents automatically. Thus, a piece of RPL plan *PL* that calls a control routine *C* and resumes its execution after *C* is done, has to call *C*, wait for the status fluent of *C* to get the value *DONE*, and then execute the rest of *PL*.

The only way RPL plans can affect the execution of control routines is by causing them to *evaporate*. RPL does not provide a command to terminate tasks. Rather, the evaporation of tasks is caused by control structures. For instance, the RPL statement (TRY-ALL P_1 ... P_n) creates subtasks for executing $P_1, \ldots, P_n$ and runs them in parallel. The successful execution of any subtask causes the TRY-ALL task to succeed and all other subtasks to evaporate. Evaporation is a dangerous concept because plans and control routines cannot be stopped at any time. A plan for crossing a street should not evaporate in the midst of its execution. Using the construct EVAP-PROTECT a programmer can postpone task evaporation until the end of such critical regions. As we will see shortly, we can use the mechanism of task evaporation to swap a new, improved plan for an old one and to terminate local planning processes if further planning is no longer beneficial.

I have considered two options for the implementation of local planning. First, I investigated extending a planning system to handle multiple planning jobs and enable it to synchronize its planning activities with the rest of the plan. Second, I sought to keep the planning system simple, solve only one planning problem at a time, but provide RPL with means to start and terminate planning processes. The second solution — providing means for creating and controlling simple planning processes — has two advantages. First, existing planning algorithms do not have to be changed a lot to use them for local planning. Second, we can employ RPL to handle the synchronization and communication between planning and execution. RPL is already designed to let the programmer specify how concurrent physical activities of a robot should interact. Thus, it is natural to extend these facilities to "mental" activities of the robot. I have implemented local planning such that it looks to the RPL interpreter as it were a physical control routine and can be used as a primitive statement in RPL control structures. By this means we can formulate expressions like the following ones as parts of structured reactive plans.

WHENEVER Event *EV* happens
    replan task *TSK* &
    execute revised plan

WHENEVER you have a better plan for *TSK*
    THEN use the better plan

Wait until you know *INFO*
    before you plan task *TSK*

7.1.1 The RUNTIME-PLAN Statement

"Anytime" algorithms [30, 53] that have the following functionality are appropriate candidates for the implementation of local planning. Given a task *T*, the algorithm tries to compute plans that are estimated to be better for performing the task *T* considering the execution of the plan so far. Each time the algorithm has found a better plan *P′* it provides *P′*, the best plan it has found so far, as its result. Several algorithms satisfying this property have been proposed for robot planning [127, 63, 116, 55].

For our implementation of local planning, we have used the variant of the XFRM planning algorithm [127, 21] that we have described in chapter 3. In XFRM planning is implemented as a search in plan space. A node in the space is a proposed plan; the initial node is a plan that represents the local planning problem. A step in the space requires three phases. First, XFRM projects a plan to generate sample execution scenarios for it. Then, in the *criticism* phase, XFRM examines these execution scenarios to estimate how good the plan is and to identify flaws in the plan by applying *critics*. A plan critic is a function that is run on a projected execution scenario and returns a set of plan failures. XFRM diagnoses the projected execution failures and

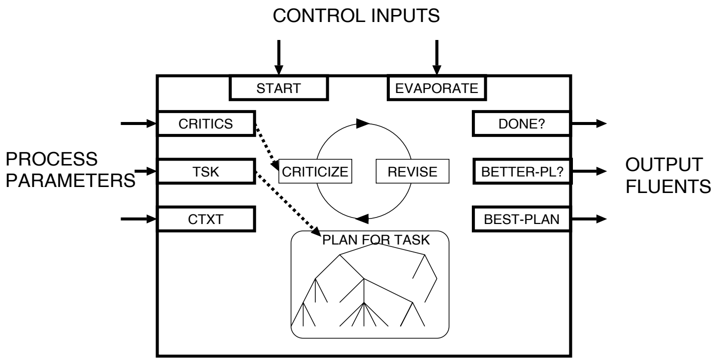


Fig. 7.1. Interface of a planning process.

uses the diagnosis to index into a set of transformation rules that are applied in the third phase, *revision*, to produce new versions of the plan, which are passed to criticism phase of the next planning cycle for evaluation. The XFRM planning algorithm can be specialized by specifying the critics to be applied.

We extend RPL with the statement (RUNTIME-PLAN *TSK CRITICS BEST-PLAN BETTER-PLAN? DONE? CTXT*), where *TSK* and *CTXT* are tasks, *CRITICS* is a set of critics, and *BEST-PLAN*, *BETTER-PLAN?*, and *DONE?* are fluents. The statement starts a planning process that tries to improve the plan for task *TSK* according to the critics *CRITICS* and provides the best plan it has found so far in *BEST-PLAN*. That is, it treats the goal *G* of task *TSK* as if it were a user command and the current plan for *TSK* as the routine plan for achieving *G* (see chapter 3). The fluent *DONE?* is set to true if the search space has been explored completely. The statement can specify to plan task *TSK* in the context of another task *CTXT*. In order to plan locally, the runtime planner extracts the subplan out of the global plan, revises it using planning, and stores the best plan it has found so far in a fluent.

A call to RUNTIME-PLAN creates a new LISP process for locally planning *TSK*. A planning process started by RUNTIME-PLAN is synchronized and communicates with the rest of the plan using the fluents described above. Viewed as a black box (see figure 7.1), the behavior of a planning process is similar to that of a control routine. It can be started and caused to evaporate. It takes the task for which a better plan is to be found and the critics to be used as arguments. The planning process automatically updates three fluents. The fluent *BETTER-PLAN?* is set true whenever the planner believes that it has found a better plan. The fluent *DONE?* is false as long as the planner is still looking for better plans. The fluent *BEST-PLAN* contains the best plan for the given task that the planner can offer in this moment.

The following example shows a typical use of local planning, how it can be started and caused to evaporate. First, fluents to be updated by the local planning process are created in the declaration part of the LET-statement. The body of the LET clause starts two parallel processes: the first one locally plans the task LOCAL-TSK, the root task of the second process. The TRY-ALL statement makes sure that the local planning process is caused to evaporate after at most 50 time units or immediately after the task LOCAL-TSK is completed.

```
(LET ((DONE? (CREATE-FLUENT 'DONE? NIL))
      (BETTER-PLAN? (CREATE-FLUENT 'BETTER-PL? NIL))
      (BEST-PLAN (CREATE-FLUENT 'BEST-PLAN NIL)))
  (PAR (TRY-ALL (WAIT-TIME 50)
                (WAIT-FOR (TASK-END LOCAL-TSK))
                (RUNTIME-PLAN LOCAL-TSK CRITICS DONE?
                                   BETTER-PL? BEST-PLAN))
       (:TAG LOCAL-TSK (WHAT-EVER))))
```

Ideally, the local planning process should reason about the local plan and the part of the global plan that is relevant for its execution. Thus, the first

```
(LET ((X 4) (Y 6))
   (WITH-POLICY
         (AVOID-OBSTACLES)
     α
     (PAR β
          (:TAG A (GO ⟨X,Y⟩))
          γ)
     δ))
```

```
(LET ((X 4) (Y 6))
   (WITH-POLICY
         (AVOID-OBSTACLES)
     (:TAG A (GO ⟨X,Y⟩))))
```

Fig. 7.2. Extraction of a local plan. (left) original plan. (right) resulting local plan. The greek letters indicate the remaining parts of the plan.

step in local planning is the construction of a plan P' that resembles the local plan in the context of the global plan closely and is much simpler than the global plan. Of course, it is impossible to construct such a P' for arbitrary reactive plans. In order to discuss this issue, let us consider the piece of RPL plan in figure 7.2 (left). This piece of plan is a LET statement that defines the local variables X and Y and initializes them to 4 and 6. The body of the LET-statement is a statement of the form (WITH-POLICY *POL BODY*) that specifies that *BODY* should be executed with *POL* as a constraint. In our example the policy is the control routine AVOID-OBSTACLES and the body contains a subplan with the name A that is supposed to get the robot to location $\langle X, Y \rangle$. When executing the piece of plan, the robot controller starts to execute the body of the WITH-POLICY statement and whenever the range sensor detects an obstacle close to the robot the policy (AVOID-OBSTACLES) interrupts the execution of the body, navigates the robot away from the obstacle and then resumes the execution of the body.

Now, suppose we want to locally replan the subplan with the tag A. There are several alternatives for determining the local plan and the relevant context of the global plan. The first one is to think only about the local plan, i.e., (ACHIEVE (LOC ROBOT ⟨X,Y⟩)). The advantage of this approach is that it is simple and cheap. The disadvantage is that when projecting the execution of the plan A, the robot might bump into obstacles, an execution failure that cannot occur because the execution of A is constrained by the policy for avoiding obstacles. The other extreme is to always project the whole plan and to discard plan revisions that are not concerned with the subplan A. The advantage of this approach is that the planner does not ignore any relevant parts of the plan; the disadvantage is that it is computationally very expensive. A third approach is to consider as relevant only the part of the global plan that is executed concurrently. This approach handles nonlocal policies, interrupts, and resources appropriately. However, it cannot handle all global effects of the local plan. For instance, a revision of a local plan to make it faster could block the execution of other concurrent subplans and thereby cause the violation of other deadlines.

Instead of trying to implement algorithms that are able to construct any conceivable planning problem for any given RPL plan we propose that plans should be designed to facilitate the construction of local planning problems. To this end, we have implemented an algorithm that makes the following assumptions about the global plan when constructing a planning problem. First, the relevant information is available at the start of the local planning process. Second, the policies surrounding the local plan and the plan specified in the context argument of RUNTIME-PLAN are the only subplans that strongly interact with the local plan. Making assumptions about plans is far more attractive than making assumptions about the robot and the world because we can engineer the plans such that the assumptions hold. In particular, transforming an RPL plan into another one that satisfies these assumptions is in most cases straightforward (see figure 4.5). An alternative is to specify a second task *CTXT* as the context of the local plan. In this case, the planner projects *CTXT*, which contains the local plan but treats the local plan as if it were a user command. That is, the planner looks only for behavior flaws caused by the local plan.

Under these assumptions a simple algorithm suffices for the construction of local planning problems. The planner constructs a local planning problem by extracting the local plan and embedding it into the policies that constrain its execution in the original plan. If a context for the local plan is specified then the plan of the context task is projected but only failures in the local plan are diagnosed and forestalled. Figure 7.2 (right) shows the plan constructed for local planning from the original in figure 7.2 (left).

7.1.2 Plan Swapping

Once an improved plan is found, the robot controller must execute it in place of the original one. We call this process *plan swapping*. The RPL statement (SWAP-PLAN *PL T*) edits the task *T* such that the plan for executing *T* is set to *PL*. To see how plan swapping can be done in RPL consider the following piece of plan that consists of two processes that are executed concurrently. The first process is a sequence of steps with some initial steps α and a final step that calls a RPL procedure FOO. The task for the final step is tagged with the name ACT. The second process consists of the local planning process and a task for swapping plans, and is caused to evaporate by the end of task ACT. The task for swapping in a new plan wakes up whenever the local planner has found a better plan (the fluent BETTER-PLAN? becomes true) and swaps in the plan stored in the fluent BEST-PLAN as the new plan for ACT after causing the old one to evaporate by editing and restarting the task ACT.

```
(PAR (SEQ α
          (:TAG ACT (FOO)))
     (TRY-ALL (:TAG PLAN
                    (RUNTIME-PLAN ACT C DONE?
                                  BETTER-PLAN? BEST-PLAN))
              (WHENEVER BETTER-PLAN?
                  (SWAP-PLAN BEST-PLAN ACT))
              (WAIT-FOR (TASK-END ACT))))
```

The plan swapping mechanism should distinguish three cases: a new plan is found (1) before starting ACT, (2) while executing ACT, or (3) after completing ACT. In the cases (1) and (3) the integration is easy. If the interpreter has not started to execute the subtask the planner simply exchanges the plan text. If the task is already performed the plan does not have to be swapped in at all. The difficult case is when the interpreter has partially executed the plan ACT. Suppose the runtime planner is to swap in a new plan for a partially executed task of the form (N-TIMES 2 *P* UNTIL *C*) after the first iteration. If *P* is picking up an object then redoing the whole statement wouldn't be a problem because the robot would in the worst case try the pickup the object more often than necessary. If however, *P* is "type in the security code of your bank card" then the robot better keeps track of how often it has already tried to type it in because the machine will eat the card if the code is typed in too often. In this case, the revised plan has to determine the parts of the original plan which have been executed and skip them. In our case, the plan can use a global counter for the remaining trials.

As for the construction of local planning problems, we assume that plans are designed to facilitate plan swapping. This assumption requires that each task evaporation leaves the robot in a physical state that is safe, and in a computational state that reflects its physical state. The first part of the assumption can be achieved by protecting critical pieces of RPL code against evaporation using the EVAP-PROTECT construct described on page 152. The second part can be satisfied in different ways. If redoing the steps that have already been executed does not do any harm no precautions have to be taken. If there are points in plan execution at which plan swapping is safe we can synchronize execution and plan swapping such that plans are swapped only at these points. For instance, in most cases it is safe to swap the body of a loop between two iterations. In the remaining cases the execution of the revised plan should ideally pick up where the original subplan stopped, which can be assured in two ways. First, the state of plan execution can be determined by sensing. In the other case the original plan has to store the state of execution in global variables and update these variables as execution proceeds. The revised plan has to decide which of its subplans to skip based on the values of these variables. We call a plan *P* *restartable* if for any task *T* that has *P* as its plan, starting the execution of *P*, causing its evaporation, and executing *P* completely afterwards causes a behavior that satisfies the task *T* (see chapter 4.2).

Most RPL plans can be made restartable using programming idioms for the corresponding RPL constructs, such as loop control structures. A typical application of a loop control structure is the manipulation of a set of objects one by one. In a first step of each iteration an object is perceived in order to obtain an object description that can be used to manipulate the object in a second step of the iteration. If such an iterative plan is caused to evaporate in the midst of the manipulation step and restarted afterwards it forgets that it is in the midst of manipulating an object and tries to start with the next object. Loops can often be made restartable, as shown below, by introducing a global variable that is set to the description of the currently manipulated object. Now, when restarted, the loop skips the perception step if the global variables contains an object description. The variable is reset immediately after the manipulation of the object is completed. Resetting is guaranteed, even in the case of evaporation, through the EVAP-PROTECT control structure.

```
(LOOP
     (IF (NOT *VAR*)
         (!= *VAR*
             (PERCEIVE-1 (λ (X) D))))
  UNTIL (NOT *VAR*)
     (EVAP-PROTECT
         (MANIPULATE *VAR*)
         (IF (DONE? ACH)
             (!= *VAR* FALSE))))
```

Thus, we can implement RPL plans such that each task *T* for which a step (SWAP-PLAN *P' T*) can be executed concurrently has a plan *P* that is restartable. Under the assumption that all plans satisfy this property, we can implement the plan swapping algorithm in a simple way: delete all the current subtasks of *T*, and create a new subtask to execute *P′*. If *T* is active, we cause all of *T*'s earlier subtasks to evaporate and start *T* anew.

```
Swap-Plan(tsk, plan)
1  ⟨$s_1, ..., s_n$⟩ ← Subtasks(tsk)
2  Delete-Subtasks($s_1, ..., s_n$)
3  Reduce(tsk, Make-Task(Execute(plan)))
4  if Partially-Executed?(tsk)
5    then Evaporate-Tasks($s_1, ..., s_n$)
6         Restart-Task(tsk)
```

7.1.3 Making Planning Assumptions

Planning assumptions are made by the planning system in order to simplify planning problems. We have added the statement (WHEN-PROJECTING *ASMP BODY*) that tells the plan projector to generate projections for *BODY* that satisfy the assumption *ASMP*. An assumption can be of the form *(SUCCEEDS TSK)*, which asserts that task *TSK* is assumed to succeed. If the projector projects *TSK* it uses a causal model that specifies the effects of a successful execution of *TSK*. If *TSK* has the form (ACHIEVE *(HAS ROBOT OB)*), the effects are not only that the robot has the object *OB* in its hand, but also that the robot is at the same location as *OB*, knows that *OB* is in its hand, that the robot's hand force sensor signals pressure in its hand, and so on.

In the implementation of robot controllers we use planning assumptions for three purposes. First, we use them to speed up local planning processes by ignoring unimportant aspects of the context of tasks. Second, we reduce the uncertainty that has to be considered by the planning system; instead of probabilistically choosing the value of a random variable the planner can explore hypothetical situations and answer questions like "how would the delivery plan work if sensing is perfect?" Finally, in order to avoid projecting local planning processes we specify the expected effects of local planning processes using planning assumptions.

Another application of planning assumptions is the assertion of the expected effects of local planning processes. We do not want the plan projector to project RUNTIME-PLAN statements for two reasons. First, projecting a RUNTIME-PLAN-statement is computationally very expensive, about as expensive as executing it. Secondly, the purpose of local planning is often to deal with unexpected information, and therefore it is unlikely that we can generate this information probabilistically. Thus, the projector ignores RUNTIME-PLAN statements and the programmer asserts the expected effects of local planning using planning assumptions.

Suppose the robot has a plan (ACHIEVE *G*) that needs to be replanned in the case of rain. Replanning can be done by wrapping a policy around the achievement step. As soon as the robot notices rain, the body of the WITH-POLICY statement is interrupted and a local planning process is started. This process runs until it finds a better plan, which is swapped in before the execution of the body is resumed. Since a global planning process is supposed to assume that the local planning process takes care of the contingency that it is raining, we have to specify a planning assumption. The planning assumption asserts that in the case of rain, the goal (ACHIEVE *G*) is guaranteed to succeed.

```
(WITH-POLICY (WHENEVER STARTS-RAINING?*
                 (RUNTIME-PLAN ACH ...)
                 (WAIT-FOR BETTER-PLAN?)
                 (SWAP-PLAN BEST-PLAN ACH))
   (WHEN-PROJECTING ((XFRM-ASSUME '(SUCCEEDS-IF-RAINING ,ACH)))
      (:TAG ACH (ACHIEVE G)))
```

The projector has to be changed minimally in order to handle planning assumptions.

```
PROJECT (TASK)
1  if TASK is assumed to be successful
2     then PROJECT TASK using abstract projection rules
3     else if TASK has form (WHEN-PLANNING
                               (XFRM-ASSUME (SUCCEEDS ?TSK))
                              ?BODY)
4             then RECORD ?TSK as assumed to be successful
5                  PROJECT(?BODY)
6             else PROJECT as normal
```

7.2 Defining Deliberative Robot Controllers

In this section we discuss several patterns of interactions between planning and execution and show how they can be implemented in extended RPL. The examples include a self-adapting iterative plan, a plan that monitors assumptions and forestalls execution failures if the assumptions are violated, and a plan that recovers from detected execution failures by replanning. These examples show the expressiveness of our extensions and how local planning can be used to make robot plans more robust.

7.2.1 Improving Iterative Plans by Local Planning

The following piece of RPL code is a loop that can adapt its behavior to changing situations by locally planning the next iteration of the loop. The loop body consists of two subplans that are performed concurrently. The first subplan controls the behavior of the robot in the current iteration. The second subplan tries to improve the plan for the next iteration. The local planning process starts with the plan for the current iteration. The planning process is caused to evaporate at the end of the current iteration. If a better plan is found, the better plan is swapped in for the next iteration.

```
(:TAG L (LOOP (!= I (+ I 1))
                  (PAR (:TAG AN-ITER (DEFAULT-ITER))
                        (LET ((NEXT-ITER (PATH-SUB ((TAGGED AN-ITER)
                                                     (ITER (+ I 1)))
                                                    L)))
                          (SEQ (TRY-ALL (SEQ (SWAP-PLAN
                                                  (TASK-PLAN AN-ITER)
                                                  NEXT-ITER)
                                              (RUNTIME-PLAN NEXT-ITER ...))
                                         (WAIT-FOR BETTER-PLAN?)
                                         (WAIT-FOR (TASK-END AN-ITER)))
                                (IF BETTER-PLAN?
                                    (SWAP-PLAN BEST-PLAN NEXT-ITER)))))))
```

Variants of this kind of planned iterative behavior have been implemented for the kitting robot [116] and for controlling the walking behavior of Ambler [150].

7.2.2 Plan Execution à la Shakey

The following RPL code piece specifies a simplified version of Shakey's [136] plan execution system PLANEX [68]. Shakey's robot control system manages a global symbolic world model that is assumed to be correct and complete. The system works as follows: given a conjunctive goal GOAL, the planning system STRIPS [69] computes a sequence of plan steps (s_1, ... s_n) that achieves GOAL when executed in the current situation, which is then executed. This is done by the procedure STRIPS that calls RUNTIME-PLAN, which updates the fluent DONE? and returns the correct plan in the variable CORRECT-PLAN.

```
(LOOP
  UNTIL (HOLDS? GOAL)
     (STRIPS GOAL CORRECT-PLAN DONE?)
     (WAIT-FOR DONE?)
     (ACTION-SEQUENCE→TRIANGLE-TABLE
        CORRECT-PLAN TRIANGLE-TABLE)
     (SWAP-PLAN TT TRIANGLE-TABLE)
     (TRY-IN-ORDER
        (LOOP
            (IF (BELIEF (LOC ROBOT ⟨?X,?Y⟩))
               (PERCEIVE-ALL (λ (O) (LOC O ⟨X,Y⟩))
                            :EXAMINE (CATEGORY COLOR SIZE)))
           (:TAG TT (TRY-IN-ORDER
                     ($c_{n-1}$ $s_n$)
                     ...
                     ($c_0$ $s_1$)
                     (T (FAIL :NO-PLAN-TO-ACHIEVE-GOAL)))
          UNTIL (HOLDS? GOAL))
        (NO-OP))))
```

In order to deal with some simple kinds of execution failures and recognize some obvious opportunities the sequence is translated into another representation. For each plan step s_i the procedure ACTION-SEQUENCE→TRIANGLE-TABLE computes the weakest condition c_{i-1} such that if c_{i-1} holds, the sequence (s_i, ... s_n) causes a state that satisfies GOAL. The new plan repeatedly perceives the current situation and executes a plan step until GOAL holds. The plan step to be executed is s_j if c_{j-1} holds and there exists no plan step s_k such that $k > j$ and c_{k-1}. If no executable step is found a failure of the type :NO-PLAN-TO-ACHIEVE-GOAL is raised and the planner computes a new plan and executes it (outer loop).[1]

7.2.3 Execution Monitoring and Replanning to Avoid Execution Failures

Since robot planners do not have perfect models of the world and their primitive robot control routines, the execution of a plan might not proceed as predicted by the planner. Usually the planner predicts for each stage of execution a set of states to be true and makes planning decisions for the later course of action that depend on some of these states. In this case, the success of future steps depends on the truth of states at earlier stages of execution. Thus, the plan can avoid future execution failures by actively sensing the truth of these states and reconsidering its course of actions if they are false. Checking these states is part of a process called *execution monitoring.*

[1] Note, the example we give does not have the full functionality of triangle tables and the program does not have the full functionality of PLANEX.

Processes which monitor vital states and signal their violation through fluents are commonly used for this kind of execution monitoring. The rest of the plan contains subplans that react to unwanted changes in the state, for instance, by replanning or reachieving the state. We can synchronize the monitoring step with the rest of the plan in many ways. For example, we could prioritize monitoring and execution, or we could stipulate that monitoring occurs only when the camera is unused, or we could restrict the temporal scope of the monitoring task, etc.

The following piece of RPL code shows a delivery plan that assumes the absence of interfering robots. A global policy is wrapped around the plan that watches out for other robots whenever the camera is not used by other tasks. If the policy detects another robot it pulses the fluent OTHER-ROBOT-AROUND*. The policy around the delivery plan replans the delivery under consideration of the possibly interfering robot whenever the fluent is pulsed.

```
(WITH-POLICY
   (LET ((PERCEIVED-ROBOTS '()))
      (LOOP (WAIT-FOR (NOT (IN-USE CAMERA)))
              (!= PERCEIVED-ROBOTS (LOOK-FOR '((CATEGORY ROBOT))))
              (IF (NOT (NULL PERCEIVED-ROBOTS))
                  (PULSE OTHER-ROBOT-AROUND*))))
   ...
   (WITH-POLICY (WHENEVER OTHER-ROBOT-AROUND*
                           (RUNTIME-PLAN DELIVERY ...))
      (:TAG DELIVERY ...)))
```

Sometimes, it is useful to implement routine plans that make assumptions that cannot be guaranteed to hold. Routine plans making such assumptions are often much more efficient and the contingencies under which they do not work might be easy to sense or very unlikely. One way to make these routine plans more robust and efficient is to install monitors that sense the robot's environment to detect violations of assumptions and opportunities, and revise the plan appropriately.

Several planners can automatically add steps for verifying the executability of subsequent parts of the plan by analyzing the causal dependency between the steps in the plan [60]. However, adding verification steps based only on the causal structure is too simplistic. Our delivery robot in the DeliveryWorld cannot and should not verify that BALL-23 is at location ⟨*10,10*⟩ while going from ⟨*0,0*⟩ to ⟨*0,3*⟩. Monitoring such states requires the robot to go different locations and compete for resources (e.g., the camera) and therefore have to be properly synchronized with the rest of the plan.

7.2.4 Recovering from Execution Failures

Sometimes execution failures cannot be forestalled and therefore local planning processes have to plan to recover from execution failures after they have occurred. Plans can often recognize execution failures, such as a missed grasping action, by checking whether the pressure measured by the hand force sensor after grasping is still zero. This execution failure together with a failure description can be generated by appending (IF (= HAND-FORCE-SENSOR* 0) (FAIL :NOTHING-GRASPED)) to the grasping routine. Failures and their descriptions are passed to supertasks and cause supertasks to fail unless they have a failure handling capability. For instance, the macro (WITH-FAILURE-HANDLER *HANDLER BODY*) can recover from a failure in *BODY* by executing *HANDLER* with the failure description as its argument. Suppose a delivery task signals a *perceptual confusion error* because more than one object has satisfied a given object description. In this case, the handler starts a local plan revision process for the delivery plan. The revised delivery plan is then executed to recover from the failure. If the failure recovery is not successful an irrecoverable failure is signalled to the super tasks.

```
(WITH-FAILURE-HANDLER
        (λ (FAILURE)
           (IF (IS perceptual-confusion-bug FAILURE)
               (TRY-IN-ORDER
                   (SEQ (RUNTIME-PLAN ERR-REC
                                 :CRITICS '(PERCEPTUAL-CONFUSION)
                                 :TIME-RESOURCE 20)
                        (:TAG ERR-REC
                              (ACHIEVE '(LOC ,DES ⟨0,8⟩)))))
                   (FAIL :IRRECOVERABLE-FAILURE FAILURE)))
               (FAIL :IRRECOVERABLE-FAILURE FAILURE)))
    (:TAG BODY (ACHIEVE '(LOC ,DES ⟨0,8⟩))))
```

There are several types of failures: failures caused by perception subplans, effector failures, and failures caused by the computational state of the structured reactive controller (for example, ambiguous or incorrect designator). Execution failures are a source of information that can be used to update the robot's model of the world. The following steps can be taken to recover from execution failures: (1) try to explain the plan failure in order to find out where the world model is wrong. (2) Take the routine plan for the subtask and plan it until it is projected to be free of flaws. (3) execute the flawless plan.

SIPE [162] and PLANEX [68] are examples of planning systems that replan after the detection of irrecoverable plan failures.

7.2.5 Some Robot Control Architectures

Since RPL is a single high-level language to coordinate planning and physical actions, we can implement a variety of robot control architectures in RPL. The implementation of the classical robot control architecture *SMPA* [136] is straightforward. It consists of a sequence of four steps. In the "sense" step the robot perceives all objects at its current location. The perception subplan updates the global variable KNOWN-THINGS* that contains designators for all objects the robot has seen so far (see chapter 2.3). In the SMPA architecture we assume that KNOWN-THINGS* constitutes a correct and complete description of the world. In the "model" step we simply set the variable KNOWN-THINGS-FOR-PLANNING* to KNOWN-THINGS*. Remember, the projector accesses the designators in KNOWN-THINGS-FOR-PLANNING* for creating the initial state of a plan projection (see chapter 3.2.2). In the "plan" step we start a planning process for constructing a plan for TOP-LEVEL-PLAN, wait until the planning process is completed, and then swap the best plan the process has found for TOP-LEVEL-PLAN.

```
(:TAG SMPA
      (LOOP (:TAG SENSE
                  (PERCEIVE-ALL (λ (OB) (OBJECT OB))
                                :EXAMINE (CATEGORY COLOR)))
            (:TAG MODEL
                  (!= KNOWN-THINGS-FOR-PLANNING*
                      KNOWN-THINGS*))
            (:TAG PLAN
                  (SEQ (RUNTIME-PLAN TOP-LEVEL-PLAN ... BEST-PLAN))
                       (SWAP-PLAN TOP-LEVEL-PLAN BEST-PLAN))
            (:TAG ACT
                  (:TAG TOP-LEVEL-PLAN (NO-OP)))))
```

A more interesting example is a simplified version of XFRM's architecture. In XFRM, planning and execution are performed in parallel. The interaction between the two processes is: whenever the planner has found a better plan it is swapped in for the one that is currently executed. This can be specified in RPL with the local planning extensions as follows. The planning process PLANNER pulses the fluent BETTER-PLAN? whenever it has found a better plan and it provides the best plan it has found so far in the fluent BEST-PLAN-SO-FAR. The controller is a loop that repeatedly executes the plan TOP-LEVEL-PLAN until it is completed. The plan TOP-LEVEL-PLAN evaporates whenever the planner has found a better plan (pulses BETTER-PLAN?). In this case the plan BEST-PLAN-SO-FAR is swapped in for the current plan for TOP-LEVEL-PLAN. The new version of TOP-LEVEL-PLAN is started from anew in the next iteration of the loop. Since TOP-LEVEL-PLAN and its revisions are restartable the revisions made by the planner are integrated smoothly in the robot's course of action.

```
(:TAG XFRM
      (PAR (:TAG PLANNER
                 (LOOP (TRY-ALL
                           (RUNTIME-PLAN
                               TOP-LEVEL-PLAN ...
                               BETTER-PLAN? BEST-PLAN-SO-FAR)
                 (WAIT-FOR PLAN-SWAPPED?))))
           (WITH-POLICY (WHENEVER BETTER-PLAN?
                              (SWAP-PLAN BEST-PLAN TOP-LEVEL-PLAN)
                              (PULSE PLAN-SWAPPED?))
              (:TAG TOP-LEVEL-PLAN ...))))
```

7.3 The Controller in the Experiment

In this section we describe the robot controller that we will use for the experiments described in the next chapter. The controller behaves as follows: given a set of top-level commands it constructs a routine plan, starts executing it, and starts a planning process for eliminating subplan interferences and other execution failures in the routine plan. Wrapped around the routine plan are policies that monitor the environment for contingencies. Whenever a contingency is detected the current planning process evaporates and a new one is started. This makes sure that the planner works on the worst flaws in the plan first. The controller always executes the best plan the planner has found so far.

The fluents used by the planning processes are defined globally. The following procedure describes the planning process that forestalls execution failures that might be caused by contingencies. The process is wait-blocked until a contingency is detected. After that it executes a loop that contains the following steps: first, updating the model used for planning and second, starting a RUNTIME-PLAN planning process. An iteration is completed then the robot has detected another contingency. This new contingency is then considered in the next iteration of the loop.

```
(DEF-INTERP-PROC ANTICIPATE-AND-FORESTALL-FAILURES (PLAN)
   (WAIT-FOR CONTINGENCY-DETECTED?*)
   (LOOP (!= KNOWN-THINGS-FOR-PLANNING*
             KNOWN-THINGS*)
         (TRY-ALL
            (SEQ (RUNTIME-PLAN
                     PLAN FALSE DONE?* BETTER-PLAN?*
                     BEST-PLAN-SO-FAR* FALSE)
                 (FAIL))
            (WAIT-FOR CONTINGENCY-DETECTED?*)))))
```

Guaranteeing that the best plan is always executed is straightforward: whenever the fluent BETTER-PLAN?* is pulsed, BEST-PLAN-SO-FAR* is swapped in for the task PLAN.

```
(DEF-INTERP-PROC EXECUTE-BEST-REACTIVE-CONTROLLER (PLAN)
   (WHENEVER BETTER-PLAN?*
      (TRY-IN-ORDER
         (SWAP-PLAN BEST-PLAN-SO-FAR* PLAN)
         (NO-OP)))))
```

The controller for the experiments in the next chapter is shown below. It consists of the structured reactive plan (SRP), two planning processes (AVOID-SUBPLAN-INTERFERENCES and CONTINGENCY-PLANNER), and the plan swapping process. These processes are executed concurrently. Three global policies, CHECK-SIGNPOSTS-WHEN-NECESSARY, CIRCUMNAVIGATE-BLOCKED-LOCATIONS, and SCAN-FOR-OTHER-ROBOTS are wrapped around the processes. The policy CHECK-SIGNPOSTS-WHEN-NECESSARY determines the robot's current location whenever it is unknown and a plan piece needs to know it.

```
(PAR (:TAG AGENT-CONTROLLER
           (WITH-POLICY (CHECK-SIGNPOSTS-WHEN-NECESSARY)
              (WITH-POLICY (CIRCUMNAVIGATE-BLOCKED-LOCATIONS)
                 (WITH-POLICY (SCAN-FOR-OTHER-ROBOTS)
                    (PAR (:TAG SRP
                               (PLAN ((:TAG MAIN
                                            (TOP-LEVEL
                                               (:TAG CMD-1 ***)
                                               ...
                                               (:TAG CMD-n ***))))))
                         (WHENEVER CONTINGENCY?*
                            (RUNTIME-PLAN
                               SRP FALSE DONE?* BETTER-PLAN?*
                               BEST-PLAN-SO-FAR* FALSE))
                         (TRY-ALL (:TAG CONTINGENCY-PLANNER
                                        (FORESTALL-FAILURES SRP))
                                  (:TAG SWAPPER
                                        (EXECUTE-CONTROLLER SRP))
                                  (WAIT-FOR (END-TASK SRP))))))))
    (:TAG ENVIRONMENT
          (TRY-ALL (:TAG OTHER-ROBOTS (RUN-OTHER-ROBOTS))
                   (WAIT-FOR (END-TASK AGENT-CONTROLLER)))))
```

7.4 Discussion

Locally planning ongoing activities will be an important capability of future service robots that perform their jobs in changing and partly unknown environments. The research reported in this paper makes important contributions to the realization of such capabilities. The main contribution is a single high-level language that coordinates planning and execution actions. To implement the language we have started with a robot control/plan language that offers powerful control abstractions for concurrent plan execution, handling asynchronous events, and causing control processes to evaporate. In addition, the language had to provide stack frames created during plan interpretation as computational objects that can be passed as arguments to local planning and plan swapping processes. We have added local planning capabilities as primitive statements that start planning processes and swap plans. The protocol for interacting with local planning processes has been designed is the same as the one for physical control routines, which enables control programs to control physical and planning actions in exactly the same way. The primitives for starting planning processes and plan swapping have built-in mechanisms for constructing local planning problems and revising partially executed plans.

Besides the language extensions we contribute solutions for two computational problems that are essential for the coordination of planning and execution: the revision of partially executed plans and the construction of local planning problems. We have shown that plans can be designed so that simple algorithms can solve these two problems robustly and efficiently.

The ideas underlying local planning of ongoing activities and the success of our realization are difficult to evaluate. Our realization of local planning required minimal extensions of an existing language for behavior-based robot control: only three statements were added. The extended control language enables plan writers to specify a wide range of very flexible interactions between planning and execution. We have seen in the previous section that, using the language, existing robot control architectures can be implemented cleanly and concisely. The examples we gave also showed that planning and physical actions have to synchronized in the same way as physical actions. Thus, the control abstractions for synchronizing physical actions are equally useful for specifying the interaction between planning and execution. In our experience, programming the interaction between planning and execution using the control structures provided by RPL made the implementation of these control architectures particularly easy.

We have successfully implemented variants of the code pieces discussed in the previous section and used them to control a simulated robot in a simulated world. In particular, we have embedded a robot controller in a simulated robot performing a varying set of complex tasks in a changing and partly unknown environment. The controller locally plans the ongoing activities of the robot. Thus whenever the robot detects a contingency, a planning process is started that projects the effects of the contingency on the robot's plan and — if necessary — revises the plan to make it more robust. Using this controller, the robot performs its jobs almost as efficiently as it would using efficient default plans, but much more reliably (see chapter 8).

Future Work.. Ultimately, we want to enable robots to reason about when to plan what and for how long. But before we can tackle these high-level reasoning problems we have to provide the basic mechanisms that allow for a flexible integration of planning and execution. These mechanisms include those for (1) turning observations of opportunities and risks into the corresponding planning problems, (2) integrating plan revisions smoothly into ongoing activities, (3) starting and controlling planning processes, and (4) synchronizing threads of planning and execution.

In our future work we want to explore techniques for reasoning about planning and acting. Although we consider program language constructs for the specification of the interaction between planning and execution is an important step towards intelligent control of robots, we also have experienced that specifying these interactions is tedious and often difficult. Ultimately, we would like the robot controller to infer both when and what portions of its plan need revision.

Related Work. Any robot that executes its plans has to coordinate planning and execution in one way or the other. Most systems implement a particular type of coordination between planning and execution. Others leave the coordination flexible. Our approach proposes to specify the interaction between planning and execution as part of the robot controllers and provides the necessary language primitives to support this.

PLANEX [68] and SIPE [162] reason to recover from execution failures. The systems first generate a complete plan and execute it afterwards. If an execution failure is detected that the execution module cannot recover from, a new plan is generated or the current plan is adapted to the new situation. Planning and action are performed in separate phases. We have sketched the RPL code for a simplified PLANEX system on page 161.

NASL [120] and IPEM [9] interleave planning and execution in order to exploit the information acquired during plan execution. NASL uses planning to execute complex actions and IPEM integrates partial-order planning and plan execution. These kinds of interactions between planning and action can be expressed in extended RPL.

$\mathcal{RS}$ [116] plans and acts in parallel but is constrained to highly repetitive tasks. It has also the ability to evaporate and revise partially executed steps but only in the context of a highly repetitive task. We have seen that variants of this kind of interaction can be implemented in RPL (see section 7.2.1).

Hybrid-layered control systems [51, 76, 31] can plan and execute in parallel by decoupling planning and execution with a sequencing layer which manages a queue of tasks. In RPL we can accomplish the same kinds of behavior: the operating system of RPL also manages a task queue which determines the urgency of tasks. The vital behavior are global policies are always active. The difference is that in RPL the task queue is not manipulated explicitly but implicitly as specified by the semantics of RPL constructs.

PRS [77] action is determined by the interaction between an intended plan and the environmental contingencies. Rational agency as the result of beliefs, intentions, and desires of the agent [32, 77]. Blackboard architectures [95] are data-driven control systems that use "meta-level" reasoning to choose between different applicable courses of action. PRS does not provide the kinds of control abstractions for various kinds of synchronizations that are provided by RPL.

Our work is closely related to TCA (Task Control Architecture) [150], which provides control concepts to synchronize different threads of control, interleave planning and execution, recover from execution failures, and monitor the environment continuously. RPL provides control structures for the same purposes at a higher level of abstraction.

8. Evaluation

This chapter discusses how the research in this book contributes to the fields of autonomous robot control and AI robot planning. We have already evaluated computational techniques used in structured reactive controllers in the chapters on transparent reactive plans, XFRML, and local planning of ongoing activities. In this chapter, we will demonstrate that taken together these techniques can improve the performance of autonomous robots considerably.

Our evaluation of structured reactive controllers consists of three steps (see, for example, [48]). First, we will argue that the computational problem that we seek to solve is an important specialization of the robot control/planning problem. We will explain how and under which circumstances programs that solve this computational problem can improve the performance of autonomous robots. Second, we will argue that structured reactive controllers can cause the robot to act competently in ways that controllers that either lack the ability of specifying flexible behavior or forestalling failures, cannot. Finally, we use the delivery robot in the DELIVERYWORLD to demonstrate that structured reactive controllers cannot only act in standard situations as time-efficiently as fixed controllers but can also handle non-standard problems that fixed controllers cannot.

8.1 Analysis of the Problem

We can state the objective of many autonomous robots as:

> *given a set of jobs, carry them out successfully without wasting resources.*

Since we use a planning system to solve the robot control problem, we refine the problem as follows.

> *Given an abstract plan P, and a description of the current situation, find an executable realization Q that maximizes some objective function V and execute it [126].*

Planning is feasible only if the planning system is equipped with the information necessary to compute (more or less precise) predictions of what will happen when the robot carries out its control program. Planning will be advantageous if the robot has to make critical control decisions that depend on the expected effects of plan execution rather than on the robot's current and prior perceptions only.

Planning as Forestalling Failures in Routine Plans The goal of this book is to design and implement a control system for a (simulated) autonomous robot with limited and noisy sensors and imperfect effectors. The control system enables the robot to accomplish varying sets of complex jobs in a partly unknown and changing world successfully without wasting resources. Therefore, we specialize the robot control problem as follows:

> *Given a set of jobs, execute the routine plan for the jobs and adapt the routine plan to avoid flaws in the robot's behavior during plan execution.*

Central to this problem statement is the notion of a routine plan, one that can accomplish its purpose in standard situations with a high expected utility. Using routine plans structured reactive controllers can

– handle standard situations without planning;
– detect non-standard situations; and
– forestall flaws that might occur when executing routine plans in non-standard situations.

What differentiates our formulation of the planning problem from many others are the ways in which the robot control system deals with information acquired during plan execution: (1) reactively, by responding immediately to sensor input and recovering from local plan failures; and (2) strategically, by revising its plan to forestall flaws.

Although forestalling flaws in routine plans yields difficult computational problems, stating the control problem this way has several important advantages:

– A planning system that revises routine plans needs less information about the current state of the world than another one that generates situation-specific plans. Since routine plans can handle standard situations, the planning system only needs to know how the current situation differs from a standard situation and can probabilistically guess the missing information assuming that everything it does not know complies with the standard.
– The robot can exploit routine aspects of its job. If the robot gets variations of the same jobs and faces similar situations over and over again, it can be equipped with a library of routine plans. Using the library, the robot does not need to generate plans from first principles for each job it gets.

To act competently autonomous robots must adapt their routine activities to avoid flaws in non-standard situations. Making the necessary decisions requires the robots to solve instances of the problem statement above. Examples of such decisions are "how to change the robot's course of action when a door is closed rather than open?" Or, "what to do if there are three objects satisfying a given description instead of just one?" Many other formulations of the robot planning problem do not help the robot to make these decisions. They assume the information at plan time to be sufficient to determine the best course of action and not to change during plan execution. Therefore, the robot has to stop and replan whenever it learns new information that might potentially be relevant for its course of action.

The anticipation and forestalling of flaws is an important class of planning operations. Classical planning relies entirely on this kind of planning operations. Classical planners anticipate and forestall two kinds of failures: the ones caused by missing plan steps and the ones caused by negative interferences between plan steps. This is sufficient because classical planning makes strong assumptions. Examples of such assumptions are, that the objects in the robot's environment have unique names that are sufficient for the robot to manipulate them, that the agent is omniscient, has perfect models of its actions, and that all changes in the environment result from the agent's own actions. Making these assumptions reduces the number of ways in which plans can fail: Assuming that objects are known and can be manipulated using their names allows the planner to neglect difficult problems in robotic sensing. Assuming omniscience excludes all failures caused by incorrect and incomplete world models. If we are willing to make strong enough assumptions — as in classical planning — we can devise small and elegant planning algorithms that compute provably correct plans.

For a planner that is embedded in an autonomous robot, however, the anticipation and forestalling of execution failures is both more general and more difficult. Embedded planners are often forced to give up unrealistic assumptions. Giving up assumptions usually means for a planner that it has to deal with additional ways in which plans may fail. A planner that does not make the assumption that the robot can refer to objects by their names will need a more detailed model of how the robot perceives objects and uses its perceptions to manipulate objects. If this model describes perception as imperfect and incomplete, the planner might have to consider that a plan might fail because the robot overlooks objects or cannot correctly perceive an object's visual properties. Also, if the robot perceives only what it needs to perform its tasks or the visual properties of objects are not unique then plans might fail because of ambiguous object descriptions, and so on. As we drop assumptions about the robot, its jobs, and its environment, robot planners have to deal with more and more kinds of failures.

8.2 Assessment of the Method

This section summarizes and analyzes structured reactive controllers, robot control systems that simultaneously execute plans and forestall predictable flaws in these plans.

8.2.1 Description of the Method

Structured reactive controllers are computational models of forestalling common flaws in autonomous robot behavior. They simultaneously execute routine plans and forestall flaws in their plans as they learn more about the world and hidden complications of their jobs. Structured reactive controllers work as follows. Given a set of jobs the planner will construct a flexible routine plan. The structured reactive controller watches out for atypical situations and possible interferences between different jobs. Planning processes predict plan failures, diagnose why they might occur, and revise the routines such that they will have a higher utility for the particular situation.

Structured reactive controllers handle standard situations by executing concurrent threads of control (monitoring surroundings and sensors while carrying out subplans), reacting immediately to sensor input (react to dropping an object), remember the objects that the robot has seen, recovering locally from execution failures (picking up an object should almost always succeed), resolving conflicts with respect to resources (no incoherent commands to the motor), and handle interruptions. They handle non-standard situations by running planning processes that predict, diagnose, and forestall the flaws in the behavior of the robot while the robot carries out its jobs.

Forestalling flaws in structured reactive plans will not work, if the robot does not have the information necessary to project informative execution scenarios, or if the rate in which the robot computes plan improvements is too slow compared to the pace in which the world changes and plans are executed.

8.2.2 Evaluation of the Method

To evaluate structured reactive controllers we compare them with other methods for controlling agents and discuss the assumptions they make, their cost, and their performance in difficult cases.

Scope of the Method. Structured reactive controllers can specify the same behaviors as behavior-based control systems. But, unlike behavior-based systems, they can also exploit information acquired while carrying out their activities by avoiding predictable flaws in their behavior. Structured reactive controllers can make informed decisions about how to change the course of action to make it robust when detecting another robot or how to accomplish a job when object descriptions in the job specifications turn out to be ambiguous.

Structured reactive controllers can also avoid the kinds of plan failures detected by classical planning algorithms (unaccomplished goal/precondition caused by a missing action or negative interference between different plan steps) and perform the respective plan transformations (insertion of new plan steps and ordering plan steps). We have described a plan revision method for introducing ordering constraints in structured reactive plans (see chapter 6.2) that is more general than its classical counterpart.

Structured reactive controllers hold a lot of promise for further development since both the language for representing plans and the notation for revising plans are very expressive. In chapter 6.3 we have shown several typical flaws in the behavior of autonomous robots that can be diagnosed and forestalled by structured reactive controllers (as, for instance, missing deadlines, flaws caused by ambiguous object descriptions). There are many other possible plan revisions that can be added to structured reactive controllers that have not been explored in this book.

Assumptions Made by Structured Reactive Controllers. Compared to other planning systems, structured reactive controllers make weaker assumptions about the capabilities of the agent, its jobs and environment. They do so at the cost of making stronger assumptions about plan representation and the kinds of information available to the planning system (routine plans that work in standard situations, models of typical flaws in the robot's behavior, and methods for forestalling the flaws).

Structured reactive controllers do not depend on perfect performances by sensors and effectors. They can deal with limited and noisy sensing. They assume, however, that perception plans produce in standard situations probably approximately accurate information. They also need probabilistic models of how the robot's sensors work in different situations. The assumption about the reliability of perception can be relaxed because structured reactive plans contain control structures that make perceptions more reliable. For example, the robot can look repeatedly until it finds what it is looking for. Or, the robot can verify its perceptions by examining objects more thoroughly. If necessary, planning processes will project the circumstances under which the robot will perceive objects and then choose sensing routines better suited for the expected situations. Structured reactive controllers deal with imperfect effector routines in similar ways: they embed them into control structures that sense the desired effects of the robot's action and recover from local execution failures. Again the controller assumes that plans, like picking up an object, have a high probability of success in standard situations and that the robot has probabilistic effector models.

Structured reactive plans neither rely on perfect information about, nor on accurate probabilistic models of, the current state of the world. They merely need to know those aspects in which the current state of the world is atypical. They project their routine plans based on a probabilistic model of what standard situations look like and on the observations the robot has made so far. Therefore, the robot does not have to sense every single aspect of the world because it might at some point become relevant for determining the best course of action. It suffices for the robot to sense just the information it needs to accomplish its jobs and detect non-standard situations. The planner is unlikely to detect behavior flaws for which it has not sensed conditions which cause the flaw with high probability.

The advantages of structured reactive controllers come at a cost: they are more expensive in terms of development costs and computational resources than other methods for autonomous robot control. The plan writer has to provide a lot of information to get the system running, most notably, routine plans for goals, causal models for behavior modules, failure models, and special-purpose plan revision methods.

– Coding a library of routine plans is a tedious and time-consuming job. Routine plans must be transparent, restartable, and work concurrently with other routine plans, even when interrupted. The XFRM framework facilitates the implementations by providing a plan language with very powerful control abstractions and declarative statements for making plans transparent.[1]
  In the long run we hope to learn libraries of routine plans automatically. Sussman's HACKER [156] is an early computer program that learns plans, called skills, for accomplishing simple block stacking problems in the "blocks world." In HACKER skill is a set of canned answer procedures indexed by problems. HACKER acquires and improves its skills by solving a sequence of increasingly difficult problems and generalizing its solutions to these problems. The idea of routine plans and HACKER's skills are very similar; the difference being that HACKER has complete and accurate knowledge of the robot's actions and the world whereas routine plans are designed to accomplish their jobs in changing and partly unknown environments. More recently, Mitchell [131, 132] and DeJong and Bennet [58] have proposed novel algorithms for learning routines for real robots.
– Another costly procedure in the development of structured reactive controllers is the specification of models of typical flaws and plan revision methods that forestall these flaws that have to be provided by a programmer. The XFRM framework supports programmers in writing these failure models and plan revision methods by providing XFRML, an expressive and concise notation for the specification of plan revision methods. In addition, the programmer can reuse parts of the failure taxonomy described in chapter 3.4.3 that contains many general failure models and plan revisions methods.
  Again, we would like the robot to learn procedures for detecting obvious flaws by just looking at the plan and the situation description without projecting and carefully analyzing the flaws. Such procedures could, for instance, detect obvious interferences between top-level commands. HACKER

[1] Other views on designing routine plans include [1, 46, 3].

[156] is able to learn methods for detecting flaws in plans. However, all these flaws are caused by violations of the "linearity assumption," the assumption that the order in which subgoals of a conjunctive goal are accomplished does not matter. Most flaws in the behavior of the delivery robot, however, are caused by ambiguous object descriptions, imperfect sensing, changes in the world caused by other robots, and so on.

– The programmer also has to equip the robot with monitoring processes that detect non-standard situations. Examples are monitors that look for empty boxes whenever the camera is not used by other processes, detect other robots, and detect situations in which more than one object matches a given object descriptions. In the XFRM framework the implementation of these monitoring processes is supported by the powerful control abstractions provided by RPL.

  Recent work in the area of robot learning has started to develop methods for learning what to monitor to keep the robot from running into trouble [158]. These learning methods take advantage of the robots' general sensors that are always active and detect when the robots are in trouble (for example, bump sensors). By learning patterns of sonar readings the robot typically receives before bumping into a closed door, the robot can develop a filter that will monitor sonar readings and detect whether the robot is likely to run into a closed door.

Limitations of Structured Reactive Controllers. Currently, the most severe limitation of structured reactive controllers is that planning processes can only project a relatively small number of random execution scenarios.[2] This reduces the number of candidate plans that can be generated and tested while a plan is executed and also limits the thoroughness with which a candidate plan can be evaluated. With the current implementation, planning processes do not produce more than three candidate plans for avoiding one particular flaw, and they project each candidate plan only two or three times.

Structured reactive controllers assume that if a particular kind of plan failure is likely to occur due to the constellation of jobs or due to a non-standard situation, the plan failure will most likely occur in the projected execution scenarios. Thus, the flaws that can be avoided by structured reactive controllers are those that are normally highly unlikely and become, under certain conditions, very likely. In addition, the robot has to detect the conditions under which the flaws become likely. We have discussed many examples of such conditions: other robots with interfering jobs, empty boxes in the neighborhood, closed doors that are normally open, and the presence of unexpected objects.

Let us analyze the impact of projecting a small number of random execution scenarios on the performance of the planning processes more carefully.

---

[2] The projection algorithm, described in [128], computes in the limit the correct distribution of execution scenarios.

For the following discussion we distinguish between two kinds of flaws: phantom flaws and predictable flaws. A phantom flaw is a flaw caused by the execution of a routine plan in a standard situation. I call this a phantom flaw because the planner will most likely not produce a better plan by revising the routine plan to avoid this flaw (even though the flaw has been projected it is very unlikely to occur). I call a flaw predictable, if a plan has a significant chance of causing the flaw in situations that satisfy a condition c and the condition c has been perceived by the robot.

When forestalling behavior flaws based on a few number of random execution scenarios, the process of anticipating and forestalling behavior flaws might have the following unwanted effects: First, the planner revises the plan to avoid a phantom flaw. Second, the planner overlooks a predictable flaw. Third, the planner forestalls a phantom flaw instead of a predictable flaw. Fourth, the planner estimates a plan with predictable behavior flaws to be better than one without such flaws.

The probability of projecting phantom flaws is quite high. Consider a routine plan that causes the robot to exhibit a behavior flaw with a probability of three percent. If we model the underlying probabilities exactly, then the planner will predict behavior flaws in standard situations in three percent of the projected execution scenarios. If the planner projects each candidate plan three times, then it might chase a phantom flaw for almost every tenth plan it generates. To avoid frequent chases of phantom flaws we use optimistic probabilistic models for the execution of sensing and action routines in standard situations.

In most cases, forestalling phantom flaws makes the behavior of the delivery robot less efficient but rarely causes behavior flaws. Typical plan revisions performed to forestall a phantom flaw might ask the robot to stabilize goals unnecessarily, or to perform its jobs in an inefficient order, or to achieve goals for more objects than necessary.

The second problem caused by forestalling behavior flaws based on few Monte Carlo projections is that the planning processes might overlook a predictable behavior flaw that is likely to occur. The probability of this happening is low (depending on the probability that the behavior flaw occurs):

	probability of overlooking flaw based on x projections	
probability of the flaw	$x = 3$	$x = 4$
50%	12.5%	6.25%
75%	1.6%	0.4%
90%	0.1%	0.01%

The probability that the planner forestalls a phantom flaw instead of a predictable flaw is also low. This is important because in this case the planner might start later to eliminate a predictable flaw and therefore install a necessary revision too late. The planner picks a phantom flaw to work on only if the severity of the phantom flaw is estimated to be at least as high as the one of the predictable flaws. Predictable flaws, however, are much more likely to occur more than once in a set of execution scenarios and therefore, their severity tends to be higher. The following table shows the probability that the planner forestalls a phantom flaw instead of the predictable flaw (assuming that the phantom flaw occurs in only one scenario and all flaws have the same severity).

	probability of picking the phantom flaw based on x projections	
probability of the flaw	$x = 3$	$x = 4$
50%	31.25%	18.75%
75%	3.95%	2.4%
90%	1.45%	0.4%

Also, because predictable behavior flaws are likely to occur more than once is it unlikely that the utility of a plan that causes predictable behavior flaws is estimated to be higher than the utility of one that causes no or only phantom flaws.

To summarize, even for a small number of randomly generated execution scenarios is it very likely that processes that anticipate and forestall behavior flaws will eliminate predictable behavior flaws. The likelihood of performing plan revisions to avoid phantom flaws can be minimized by using optimistic probabilistic models for the execution of sensing and action routines in standard situations.

Increasing the number of execution scenarios that can be generated and analyzed in the available computation time will result in an increase in the range of flaws that structured reactive controllers are able to avoid. We have several options for increasing the number of execution scenarios. Besides speeding up the temporal projection algorithm, there are some additional very promising techniques that enable the structured reactive controllers to perform more projections. For instance, the structured reactive controller can constrain planning to subplans or project plans in hypothetical situations (see chapter 7).

Performance in Difficult Cases. An issue we have not discussed yet is the performance of structured reactive plans in difficult situations such as situations in which the planning process cannot detect a flaw. If a flaw cannot be detected the structured reactive controller will continue executing the default plans. If the planning system cannot determine the exact cause of a flaw the planning process will apply more general plan revision methods which are less likely to avoid the flaw. If the planning processes produce plan revisions after the corresponding parts of the plan are executed, these revisions will not affect the course of action taken by the robot.

Sometimes the revisions performed by structured reactive controllers make the behavior of the robot worse. Suppose the robot has to get the blue ball at location ⟨*2,2*⟩. There is a blue and a green ball at ⟨*2,2*⟩. When the robot looks for the blue ball it finds the blue ball and also mistakes the green ball for a blue ball. It detects its perceptual confusion and will revise its plan to avoid it. The resulting plan might be worse than the original one, even though it is better in the robot's mind.

8.3 Demonstration of the Method in the DeliveryWorld

So far we have analyzed structured reactive plans analytically. A good metric for evaluating the practical usefulness of structured reactive controllers is the amount by which they can improve the performance of the robots they control [166, 47, 138]. The overall performance summarizes several measures such as the quality of the plan revisions, the time resources needed to compute these revisions, how robust plan revision is, and so on. The performance of the robot is measured in terms of how well the robot accomplishes its jobs and how long it takes to accomplish the jobs.

This section describes three kinds of experiments we have performed to verify that structured reactive controllers can improve the behavior of the delivery robot in the DeliveryWorld. The first experiment compares the time resources required by structured reactive controllers with those consumed by fixed routine plans in standard situations. The second experiment explores how well structured reactive controllers can do if they have the necessary information and are not under time pressure. The third experiment describes problem instances that can be handled competently by structured reactive controllers but not by fixed routine plans.

8.3.1 Evaluating SRCs in Standard Situations

The first experiment compares the performance of routine plans and structured reactive controllers in situations that do not require any planning. The experiment is designed as follows. We define a probability distribution on sets of jobs and situations and draw ten random problem instances from the distribution. Figure 8.1 shows the two controllers we use. The routine plan deals with common problems like balls slipping out of the robot's gripper by trying to pick up objects repeatedly until the grasp is successful. The structured reactive controller (see chapter 7.3 for details) starts with the default plan but tries to forestall failures.

The experiment compares the performance of the routine plan and the structured reactive controller. Since both controllers specify non-deterministic behavior, we run each controller fifty times on each problem in order to get performance profiles for them.

The expected result is that routine plans and structured reactive controllers should exhibit non-deterministically the same behavior and therefore

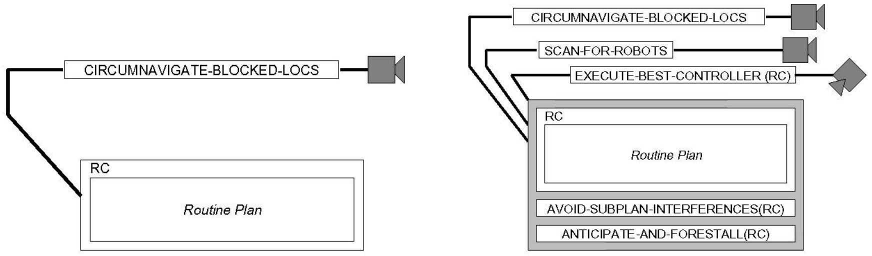

CIRCUMNAVIGATE-BLOCKED-LOCS
RC
Routine Plan
CIRCUMNAVIGATE-BLOCKED-LOCS
SCAN-FOR-ROBOTS
EXECUTE-BEST-CONTROLLER (RC)
RC
Routine Plan
AVOID-SUBPLAN-INTERFERENCES(RC)
ANTICIPATE-AND-FORESTALL(RC)

Fig. 8.1. The configuration of the SRC used in the experiments performed in this chapter.

their performance profiles in standard situations should be similar. The only time that planning interacts with plan execution is when the planning process has found a better plan and the plan swapping process swaps the revised subplans into the plan that is executed. However, in standard situations routine plans are flawless and thus planning should not interfere with the execution of the routine plan. It is still important to verify the performance in standard situations to check for the correctness of the implementation.

We compare routine plans and structured reactive controllers with respect to a particular probability distribution of problem instances. A problem instance consists of a set of jobs and an initial situation. The random problem generator creates sets of commands that contain between one and five top-level commands. The jobs are either delivery or painting jobs and may ask the robot to accomplish the job for a single or a set of objects. The descriptions of the objects for which the goals are to be achieved and the goals are also randomly generated.

The initial situation is generated based on the set of top-level commands. For universally quantified goals the random generator for the initial situation creates between zero and five objects for which the goal is to be achieved. For an ACHIEVE-command it generates between zero and two objects matching the designator in the command. Besides the objects for which the commands are to be accomplished the problem generator creates additional objects and spreads them in the world.

The most complex of the ten randomly generated problem instances contains the commands

```
CMD-1:  (ACHIEVE-FOR-ALL (λ (X) (AND (LOC X ⟨1,13⟩)
                                     (CATEGORY X BLOCK)
                                     (COLOR X BLACK)))
                         (λ (X) (LOC X ⟨1,4⟩)))
CMD-2:  (ACHIEVE-FOR-ALL (λ (X) (AND (LOC X ⟨1,13⟩)
                                     (CATEGORY X BOX)))
                         (λ (X) (LOC X ⟨4,9⟩)))
CMD-3:  (ACHIEVE-FOR-ALL (λ (X) (AND (LOC X ⟨1,4⟩)
                                     (CATEGORY X PYRAMID)
                                     (NOT (COLOR X BLACK))))
                         (λ (X) (COLOR X BLACK)))
CMD-4:  (ACHIEVE ((λ (X) (LOC X ⟨4,3⟩)))
                 (CREATE-DESIG "Desig from Problemdef"
                               '((X-COORD 4) (Y-COORD 9)
                                 (CATEGORY BLOCK)
                                 (COLOR WHITE)))),
```

which are to be accomplished in an initial state that contains among many others the following objects

```
(!= CMD-RELATED-OBJECTS*
    '((OBJECT BLOCK ((COLOR BLACK) (TEXTURE PLAIN)) 1 13)
      (OBJECT BLOCK ((COLOR BLACK) (TEXTURE PLAIN)) 1 13)
      (OBJECT BLOCK ((COLOR BLACK) (TEXTURE PLAIN)) 1 13)
      (OBJECT BOX ((COLOR BLACK) (TEXTURE PLAIN)) 1 13)
      (OBJECT BOX ((COLOR LIGHT-GRAY) (TEXTURE PLAIN)) 1 13)
      (OBJECT BOX ((COLOR GRAY) (TEXTURE CHECKED)) 1 13)
      (OBJECT BOX ((COLOR DARK-GRAY) (TEXTURE PLAIN)) 1 13)
      (OBJECT PYRAMID ((COLOR LIGHT-GRAY) (TEXTURE PLAIN)) 1 4)
      (OBJECT PYRAMID ((COLOR DARK-GRAY) (TEXTURE PLAIN)) 1 4)
      (OBJECT PYRAMID ((COLOR DARK-GRAY) (TEXTURE CHECKED)) 1 4)
      (OBJECT BLOCK ((COLOR WHITE) (TEXTURE PLAIN)) 4 9))).
```

The delivery robot starts at location ⟨*0,13*⟩. No exogenous events happen and no other robots change the world while the robot solves this problem instance.

```
(DEFINE-WORLD-STATE
    :THINGS (APPEND BG-OBJECTS* FG-OBJECTS*)
    :OTHER-ROBOTS '()
    :EVENTS '()
    :OTHER-ROBOTS '()
    :OWN-POSITION (QUOTE (0 13)))
```

Initially, the robot knows some of the objects in the world and its own location.

```
(DEFINE-AGENT-STATE
    :KNOWN-THINGS '((OBJECT BOX ((COLOR DARK-GRAY)
                                  (TEXTURE PLAIN))
                            1 8)
                    ...)
    :KNOWN-THINGS-FOR-PL '((OBJECT BLOCK ((COLOR BLACK)
                                          (TEXTURE PLAIN))
                                   1 13)
                           ...)
    :KNOWN-EVENTS '()
    :KNOWN-ROBOTS '()
    :OWN-POSITION '(0 13))
```

The experimental result that measures the time consumed to accomplish the jobs is shown in the following table. Both controllers accomplished their jobs completely.

	fixed	**planning**
data points	50	50
low	1438	1505
high	1707	1627
mean	1548.6	1537.7
median	1541.5	1536.0
std. deviation	43.9	25.4

Over the ten problem instances the difference between the mean of the time resources consumed by the routine plan and the structured reactive controller did not differ more than 0.6 percent, the standard deviation not more than 1.2 percent. The difference in the least amount of time required to solve a problem instance did not differ more than 4.3 percent (average is less than 1.4 percent). The difference in the maximal amount of time needed was smaller than 9.5 percent (average was less than 2.3 percent).

8.3.2 Comparing SRCs with the Appropriate Fixed Controller

The second experiment compares structured reactive controllers with several other controllers:

– The cautious plan. The *cautious* plan for a specific contingency c always acts as if interference caused by c is probable. For the contingency that objects disappear from location $\langle 0,10 \rangle$, the cautious plan will nail the balls down immediately after their delivery. In the case that balls might change their shapes, the ultracautious plan will mark each ball and then deliver each marked object. But if the planner has knowledge that interference is not currently likely, the cautious plan will do a lot of unnecessary work.
– The routine plan.

- The structured reactive controller. The structured reactive controller is equipped with a model of the contingencies that predicts accurately whether and which the contingency occurs and forestalls possible flaws while carrying out its routine plan. The structured reactive controller is also equipped with enough computation time.
- The clairvoyant controller. The clairvoyant, uses a perfect oracle to choose the best course of action. The controller is equipped with the routine plan and an appropriate plan for each contingency tested in the experiment. Thus as its first step the controller chooses the appropriate plan for the problem instance and then executes it.

The desired result in this experiment is that if the contingency does not occur, the structured reactive controller should perform as well as the clairvoyant controller and better than the cautious plan. If a contingency does occur the robot controlled by the structured reactive controller should behave almost as well as the clairvoyant controller and better than the routine plan.

We have performed the following experiment: the robot is at ⟨*0,0*⟩ and has to get all balls from ⟨*0,8*⟩ to ⟨*0,10*⟩. While carrying out the task, the following contingencies may or may not occur: balls may disappear or change their shape, sensor programs may overlook objects or examine them incorrectly.

For each contingency c we compare the performance of the four robot controllers. We run each controller in two scenarios: in one the contingency c occurs and in the other it does not. Figure 8.2 shows the trials for four

command	time	goals	utility	time	goals	utility
	balls disappear			balls don't disappear		
routine	241	2/4	159.8	237	4/4	360.5
cautious	323	4/4	306.1	325	4/4	305.8
planned	347	4/4	302.1	248	4/4	358.6
clairvoyant	323	4/4	306.1	237	4/4	360.5
	balls change their shape			balls don't change		
routine	153	2/4	174.5	246	4/4	359.0
cautious	315	4/4	347.5	305	4/4	349.2
planned	325	4/4	345.8	251	4/4	358.2
clairvoyant	315	4/4	347.5	246	4/4	359.0
	balls are overlooked			balls aren't overlooked		
routine	200	3/4	266.6	240	4/4	360.0
planned	255	4/4	357.5	252	4/4	358.0
clairvoyant	240	4/4	360.0	240	4/4	360.0

Fig. 8.2. Comparison between routine plan, cautious plan, clairvoyant plan, and planning simultaneous with execution of the routine plan. Duration is the time needed to perform the task in seconds. achieved goals specifies the number of goals that have been achieved. Benefit is the amount of "utility" earned by the robot, where 10 minutes execution time cost 100 units, a delivered ball brings 100 units if it is not nailed down, 90 units otherwise.

contingencies. As expected, the cautious plans are significantly slower than the routine plan but achieve all their subgoals whether or not c occurs. The routine plan works well only if c does not occur; if c does occur, some subgoals are not achieved successfully. The controller with planning is as robust as the cautious plan. It is also on average 12.8 seconds (less than 5%) slower than the executing the best plan that achieves all the goals on a given trial. Planning costs are this low because XFRM plans during the idle time in which the interpreter was waiting for feedback from control routines. In terms of average benefit, the controller assisted by XFRM was better (337.0) than the cautious (321.0) and the routine plan (261.7).

We should interpret the experimental results carefully: the structured reactive controller had a very good model of contingencies. However, we believe that the experiment shows that structured reactive controllers can outperform robot controllers that do not plan and be almost as efficient as a system that knows the best plan to use on each occasion.

It might seem that there is an alternative we did not test, namely, to write a careful reactive plan that tests its world model once, then switches to a cautious version if interference is anticipated. There are two main reasons to reject this approach: (1) It is not clear what to test; the interference may be the result of several interacting processes that emerge in the projection, so that the simplest test might just *be* the projection itself. (2) The transformation rules are not specific to a particular plan. Rather than write alternative versions of every plan, we can write a single transformation that can revise a variety of plans to compensate for certain standard bugs.

8.3.3 Problems that Require SRCs

In the following and last series of experiments we test the performance of the delivery robot in the DELIVERYWORLD in problem instances that challenge the capabilities of structured reactive controllers. The problem instances are carefully chosen to bring about flaws in the behavior of the robot that are caused by the routine plans. Originally we wanted to randomly generate a set of problem instances. However, we decided not do this for several reasons: For a world like the DELIVERYWORLD we would have had to tune the probability distribution significantly to generate such problem instances frequently. For instance, to avoid interference with another robot, the delivery robot must be in the neighborhood of the delivery robot some time before the flaw would occur and must later cause a very specific effect at a particular location. We have to bias the problem generator even more to get different kinds of contingent situations. Finally, it is not of great interest to us to show that structured reactive controllers can, for a specifically designed problem distribution, outperform fixed robot controllers. We rather want to demonstrate that structured reactive controllers can handle important kinds of problems that cannot handled by other controllers.

We have therefore chosen to make up problem instances that are characteristic for important classes of contingencies and will challenge the capabilities of structured reactive controllers. Thus, in this experiment we will show that by adding the appropriate failure models and plan revision methods to a structured reactive controller an autonomous robot can handle more and more situations competently. It could not do so without structured reactive controllers.

Ambiguous object descriptions. Through the application of structured reactive controllers autonomous robots gain the capability of handling ambiguous object descriptions more competently. Consider our first experiment: the robot is asked to deliver a particular ball BALL-27 to location ⟨*0,12*⟩. To issue this command we provide a description of BALL-27 and specify that the description designates a particular object:

```
(!= BLACK-BALL*
    (CREATE-DESIG "Ball"
                   '((CATEGORY BALL) (COLOR BLACK) (X-COORD 0)
                     (Y-COORD 10))))

(CORRESPONDS BLACK-BALL* BALL-27)

(TOP-LEVEL
    (:TAG COMMAND-1
          *(ACHIEVE ((λ (B) (LOC B ⟨0,12⟩))
                     BLACK-BALL*))))
```

When the robot looks for the black ball it will find three black balls at location ⟨*0,10*⟩. The routine plan unifies one of the balls with the designator given in the top-level command and starts delivering it. At the same time local planning is triggered to deal with this unexpected situation. When projecting the plan in the situation where three balls are at location ⟨*0,8*⟩ the planning process detects that sometimes the plan fails to deliver BALL-27. It identifies perceptual confusion as the cause of the flaw and applies the plan revision "generalize achievement goal" to the plan (the failure model and the plan revision rule are discussed in chapter 6.3.1). Thus, while continuing the delivery of the ball, the robot performs the following plan revision.

```
(TOP-LEVEL
    (:TAG COMMAND-1
          (REDUCE *(ACHIEVE ((λ (B) (LOC B ⟨0,10⟩)) BLACK-BALL*))
                  (SEQ *(ACHIEVE ((λ (B) (LOC B ⟨0,12⟩)) BLACK-BALL*))
                       (ACHIEVE-FOR-ALL
                           (λ (DES-4)
                              (AND (CATEGORY DES-4 BALL)
                                   (COLOR DES-4 BLACK)
                                   (LOC DES-4 ⟨0,10⟩)))
                           (λ (BLACK-BALL-7)
                              (LOC BLACK-BALL-7 ⟨0,12⟩))))
                  GENERALIZE-GOAL)))
```

The structured reactive controller accomplishes the job reliably (in all ten trials we have run) while the routine plan has only a 33% chance of being successful.

If there were many black balls at location ⟨*0,10*⟩, the planning process would revise *(ACHIEVE ((λ (B) (LOC B ⟨0,10⟩)) BLACK-BALL*)) into (NO-OP), the plan that does nothing. Even though the generalized plan accomplished the job, the time costs exceed the benefits. When the particular object does not matter, the structured reactive controller will keep executing the routine plan.

The robot will have additional options for dealing with ambiguous object descriptions, if the ambiguity is caused by other subplans. Thus, in another experiment the robot must accomplish a second job

```
(ACHIEVE-FOR-ALL (λ (X) (AND (CATEGORY X BALL)
                             (LOC X ⟨0,8⟩)))
                 (λ (X) (LOC X ⟨0,10⟩)))
```

and BALL-27 is the only black ball at location ⟨*0,10*⟩ at the beginning of plan execution. Now the planning process finds that the ambiguity is caused by the subplan for command COMMAND-2 and revises the plan as follows:

```
(PARTIAL-ORDER
   ((:TAG MAIN
          (TOP-LEVEL
              (:TAG COMMAND-1
                    (REDUCE
                        (ACHIEVE ((λ (B) (LOC B ⟨0,12⟩)) BLACK-BALL*))
                        (LET (...)
                            ...
                            (:TAG PERCEIVER-5 ...)
                            ...)))
              (:TAG COMMAND-2
                    (:TAG CONFUSER-6
                          (ACHIEVE-FOR-ALL
                               (λ (X) (AND (CATEGORY X BALL)
                                           (LOC X ⟨0,8⟩)))
                               (λ (X) (LOC X ⟨0,10⟩))))))))
   (:ORDER PERCEIVER-5 CONFUSER-6))))
```

In the experiment we placed three black balls at location ⟨*0,8*⟩. Using the routine plan the delivery robot has a less than 50% chance of accomplishing both jobs successfully. The probability that the structured reactive controller will accomplish both jobs is about 95% and the likelihood that it will not discover the possible interference is only about 12% (when projecting each candidate plan three times). In ten trials, the structured reactive controller has accomplished both jobs ten times whereas the routine plan succeeded only four times.

Any robot controller that uses object descriptions to manipulate particular objects is likely to run into problems caused by ambiguous and incorrect object descriptions. To accomplish its commands competently despite ambiguous object descriptions, the robot must predict what will happen if it deals with the ambiguities in one way or the other and must choose the option with the highest expected utility. In order to detect ambiguities the planning process needs to reconstruct the value of a data structure at a particular point during plan execution and compare it with the state of the world. The plan revisions in the experiments above also made use of RPL's control abstractions. To summarize, acting competently in the experiments above requires the robot to have the capabilities provided by structured reactive controllers.

Dealing with other robots. Autonomous robots with structured reactive controllers can also act competently in environments that are populated by more than one robot. In the following experiments the delivery robot will detect other robots in its neighborhood and learn about their jobs. The delivery robot will project its plan in the context of the other robots to detect flaws in its own behavior that might be caused by the other robots. If necessary, the delivery robot will revise its plan to make it more robust in the presence of the other robot.

In the subsequent experiment the delivery robot must get all balls from location ⟨*0,8*⟩ to ⟨*0,10*⟩. While going to location ⟨*0,8*⟩ the robot senses another robot that is to change the shapes of objects at location ⟨*0,8*⟩.

```
(PARTIAL-ORDER
   ((:TAG MAIN
         (TOP-LEVEL
            (:TAG COMMAND-2
                 (REDUCE
                    (ACHIEVE-FOR-ALL (λ (X)
                                        (AND (CATEGORY X BALL)
                                             (LOC X ⟨0,8⟩)))
                                     (λ (X) (LOC X ⟨0,10⟩)))
                    ...))))))
```

When projecting its routine plan the delivery robot predicts that it will not deliver some of the objects it is supposed to deliver because it will not recognize them once the other robot has changed their shapes. In order to make its behavior more robust, the delivery robot changes its plan as follows. It first goes to location ⟨*0,8*⟩ and marks all balls with a particular label MARK-3 and changes the delivery routine such that it delivers each object at location ⟨*0,8*⟩ that has the label MARK-3 instead of delivering all balls.

```
(PARTIAL-ORDER
   ((:TAG MARK-CHANGING-OBJS-3
          (ACHIEVE-FOR-ALL (λ (X) (AND (MARK X NONE)
                                       (CATEGORY X BALL)
                                       (LOC X ⟨0,8⟩)))
                           (λ (X) (MARK X MARK-3))))
    (:TAG MAIN
          (TOP-LEVEL
             (:TAG COMMAND-2
                   (REDUCE
                      (ACHIEVE-FOR-ALL
                         (λ (X) (AND (CATEGORY X BALL)
                                     (LOC X ⟨0,8⟩)))
                         (λ (X) (LOC X ⟨0,10⟩)))
                      (ACHIEVE-FOR-ALL
                         (λ (X) (AND (LOC X ⟨0,8⟩)
                                     (MARK X MARK-3)))
                         (λ (X) (LOC X ⟨0,10⟩)))
                     MARK-CHANGING-OBJECTS)))))
   (:ORDER MARK-CHANGING-OBJS-3 MAIN MARK-CHANGING-OBJECTS))))
```

The extent of improvement achieved by the structured reactive controller depends on how likely the interference is, how fast the delivery robot can install and execute the revision, and the pace in which the other robot changes objects. In our experiments the delivery robot controlled by the structured reactive controller delivered all three instead of only two balls as it did controlled by routine plan.

An alternative is to make sure that each object that is a ball does not change its shape. This revision has flaws if the other robot changes objects into balls. Then the revised plan will deliver objects it does not have to deliver and waste time.

```
(PARTIAL-ORDER
   ((:TAG PREVENT-CHANGING-OBJS-2
          (STABILIZE-FOR-ALL (λ (X) (AND (CATEGORY X BALL)
                                         (LOC X ⟨0,8⟩)))
                             (λ (X) (CATEGORY X BALL))))
    (:TAG MAIN
          (TOP-LEVEL
             (:TAG COMMAND-1
                   (REDUCE
                      (ACHIEVE-FOR-ALL
                         (λ (X) (AND (CATEGORY X BALL)
                                     (LOC X ⟨0,8⟩)))
                         (λ (X) (LOC X ⟨0,10⟩)))
                      ...)))))
   (:ORDER PREVENT-CHANGING-OBJS-2 MAIN))))
```

In another experiment the delivery robot senses another robot that tells the delivery robot that its job is to remove objects from location ⟨*0,10*⟩. Because ⟨*0,10*⟩ is the destination of the delivery job the delivery robot projects that balls might disappear after their delivery. In this situation the robot changes its plan and makes sure that the balls do not leave their destination after being delivered. This revision is described in more detail in chapter 6.2.

```
(PARTIAL-ORDER
   ((:TAG MAIN
         (TOP-LEVEL
             (:TAG COMMAND-3
                   (REDUCE
                      (ACHIEVE-FOR-ALL (λ (X) (AND (CATEGORY X BALL)
                                                  (LOC X ⟨0,8⟩)))
                                       (λ (X) (LOC X ⟨0,10⟩)))
```

```
(PARTIAL-ORDER
   ((:TAG MAIN
         (TOP-LEVEL
             (:TAG COMMAND-3
                   (REDUCE
                      (ACHIEVE-FOR-ALL (λ (X) (AND (CATEGORY X BALL)
                                                  (LOC X ⟨0,8⟩)))
                                       (λ (X) (LOC X ⟨0,10⟩)))
                      (LOOP
                         (PROCESS SINGLE-DELIVERY
                            ...
                            (REDUCE
                               *(ACHIEVE ((λ (X) (LOC X ⟨0,10⟩))
                                          CURR-GOAL-VAR-3))
                               (SEQ *(ACHIEVE ((λ (X) (LOC X ⟨0,10⟩))
                                               CURR-GOAL-VAR-3))
                                    *(STABILIZE ((λ (X) (LOC X ⟨0,10⟩))
                                                 CURR-GOAL-VAR-3)))
                               CLOBBERED-BY-EXT-EV.REV-1)
                            ...)
                         UNTIL (NULL CURR-GOAL-VAR-3)
                         (!= CURR-GOAL-VAR-3 FALSE))))))))
```

In the experiment the structured reactive controller improves the behavior of the delivery robot such that the robot accomplishes that all three balls will be at their destination upon plan completion (in all ten trials). When the robot is controlled by the routine plan none of the balls stay at their destination till the plan is completed (in all ten trials).

Revising the course of action in order to avoid interferences between the activities of different robots is an important capability cooperative environments. Again the revisions performed make use of RPL's control structures and reason about the process of plan interpretation as well as the physical effects of the plan.

Subplan Interferences. The next experiment shows the ability of structured reactive controllers in handling interferences between different subplans. If an already achieved goal is clobbered by the robot's own action, then introduce an ordering on subplans such that the clobbering action is necessarily executed before the clobbered subgoal is being achieved. We have seen how the robot avoids interferences between subplans in chapter 6.2.

```
(PARTIAL-ORDER
   ((:TAG MAIN
          (TOP-LEVEL
             (:TAG COMMAND-5
                   (REDUCE
                      (ACHIEVE-FOR-ALL
                          (λ (X) (AND (CATEGORY X BALL)
                                      (LOC X ⟨0,10⟩)))
                          (λ (X) (LOC X ⟨2,10⟩)))
                      ...))
             (:TAG COMMAND-4
                   (REDUCE
                      (ACHIEVE-FOR-ALL
                          (λ (X) (AND (CATEGORY X BALL)
                                      (LOC X ⟨0,8⟩)))
                          (λ (X) (LOC X ⟨0,10⟩)))
                      ...))))))
```

```
(PARTIAL-ORDER
   ((:TAG MAIN
          (TOP-LEVEL
             (:TAG COMMAND-5
                   (REDUCE
                      (ACHIEVE-FOR-ALL
                          (λ (X) (AND (CATEGORY X BALL)
                                      (LOC X ⟨0,10⟩)))
                          (λ (X) (LOC X ⟨2,10⟩)))
                      (:TAG VIOLATOR-1
                          (LOOP ...))
             (:TAG COMMAND-4
                   (:TAG PROTECTEE-0
                         (REDUCE
                            (ACHIEVE-FOR-ALL
                                (λ (X) (AND (CATEGORY X BALL)
                                            (LOC X ⟨0,8⟩)))
                                (λ (X) (LOC X ⟨0,10⟩)))
                            ...))))))))
   (:ORDER VIOLATOR-1 PROTECTEE-0))
```

In the experiment we have placed two balls at ⟨*0,8*⟩ and two balls at ⟨*0,10*⟩. The routine plan has a fifty percent chance of delivering three balls and a less than nine percent chance of delivering all four balls correctly. In the remaining cases the delivery robot delivers only two balls correctly. The delivery robot controlled by the structured reactive controller delivers all balls correctly.

Adding ordering constraints to a course of action is arguably the most important operation that planning systems perform. This planning operation is more general than the corresponding operation of other planning systems because structured reactive controllers add ordering constraints between subplans in plans with complex internal structure and various kinds of control structures.

Handling deadlines. Another ability an autonomous robot can gain by employing structured reactive controller is a more flexible handling of deadlines for top-level commands. Consider the following experiment. The robot is given to jobs; one of them has a deadline.

```
(!= CMD1* (ACHIEVE-FOR-ALL (λ (X) (AND (CATEGORY X BALL)
                                       (LOC X ⟨0,8⟩)))
                           (λ (X) (LOC X ⟨0,10⟩))))
(!= CMD2* ((ACHIEVE-FOR-ALL (λ (X) (AND (CATEGORY X BALL)
                                        (LOC X ⟨0,11⟩)))
                            (λ (X) (LOC X ⟨2,10⟩)))
           :DEADLINE 220))
```

In this experiment the delivery robot has revised its routine plan such that it executes the command with the deadline as the first one. Thereby the delivery robot has successfully accomplished both jobs without missing the deadline.

```
(PARTIAL-ORDER
   ((:TAG MAIN
          (TOP-LEVEL
              (:TAG COMMAND-2
                    (REDUCE
                       (ACHIEVE-FOR-ALL
                            (λ (X) (AND (CATEGORY X BALL)
                                        (LOC X ⟨0,11⟩)))
                            (λ (X) (LOC X ⟨2,10⟩)))
                       ...))
              (:TAG COMMAND-1
                    (REDUCE
                       (ACHIEVE-FOR-ALL
                            (λ (X) (AND (CATEGORY X BALL)
                                        (LOC X ⟨0,8⟩)))
                            (λ (X) (LOC X ⟨0,10⟩)))
                       ...)))))
    (:ORDER COMMAND-2 COMMAND-1 PROMOTE-TIME-CRITICAL-TLC))))
```

Overlooking Objects. In our last experiment we see what the delivery robot did when it got the job to get all balls from location ⟨*0,8*⟩ to location ⟨*0,10*⟩ and learned that location ⟨*0,8*⟩ is wet. The structured reactive controller revised the perception subplan such that it uses the sensing strategy WEAK-CONSTRAINT-SENSOR instead of the default sensing strategy. The revised sensing strategy is less likely to overlook objects but more likely to classify objects that are not balls as balls.

```
(PARTIAL-ORDER
    ((:TAG MAIN
            (TOP-LEVEL
             (:TAG COMMAND-2
                    (REDUCE
                     (ACHIEVE-FOR-ALL
                           (λ (X) (AND (CATEGORY X BALL)
                                       (COLOR X BLACK)
                                       (LOC X ⟨0,8⟩)))
                           (λ (X) (LOC X ⟨0,10⟩)))
                     (LOOP
                      (PROCESS SINGLE-DELIVERY
                       ...
                       (IF (NULL CURR-GOAL-VAR-3)
                           (!= CURR-GOAL-VAR-3
                               (REDUCE
                                   *(PERCEIVE-1
                                       '(λ (X) (AND (CATEGORY X BALL)
                                                    (LOC X ⟨0,8⟩))))
                                   (WITH-SENSING-ROUTINE
                                           WEAK-CONSTRAINT-SENSOR
                                       *(PERCEIVE-1
                                         '(λ (X) (AND (CATEGORY X BALL)
                                                      (LOC X ⟨0,8⟩)))))
                                   OVERLOOKED-OBJECT.REV-1)))
                       ...))))))))
```

8.4 Related Work

We will now relate structured reactive controllers to work in other areas of autonomous robot control: architectures for autonomous robot control, robot control/plan languages, and robot planning.

8.4.1 Control Architectures for Competent Physical Agents

A robot control architecture is a programming framework for specifying sensor and effector routines, and control programs that activate and deactivate these routines to accomplish their jobs. There are various dimensions to the problem of controlling the problem-solving behavior of physical agents. These dimensions include the amount of information available to the robot, the time available for selecting an action, the number and variations of jobs the robot has to accomplish.

A dimension that discriminates the most important categories of robot control architectures well is the degree to which the complexity of the control problem is caused by the environment or by the jobs to be accomplished. Figure 8.3 shows different types of control architectures ordered along this dimension. At the left end of the spectrum are control systems, which try to model the environment at a very detailed level using complex mathematical

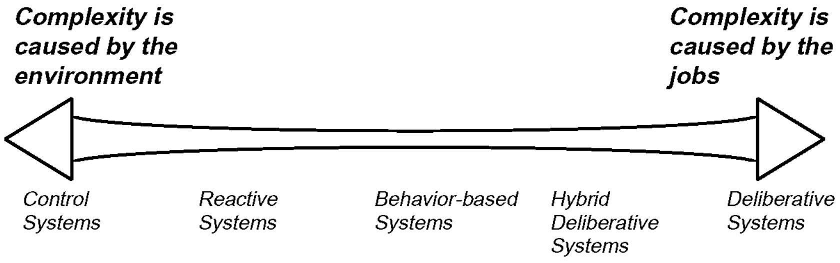

Complexity is caused by the environment
Complexity is caused by the jobs
Control Systems
Reactive Systems
Behavior-based Systems
Hybrid Deliberative Systems
Deliberative Systems

Fig. 8.3. Spectrum of causes for the complexity of the robot control problem.

models like differential equations [137, 56]. At the right end of the spectrum are classical planning architectures, which make strong simplifying assumptions about the environment but reason very carefully about how jobs can be accomplished. In between are reactive, behavior-based, and hybrid deliberative systems.

Structured reactive controllers are hybrid deliberative systems that share many characteristics and capabilities with behavior-based systems. The reason that structured reactive systems are an important category of control systems in this spectrum is that their strength is accomplishing daily jobs in cooperative environments, like offices and factories — the kinds of jobs and environments humans perform well in.

Situated Robot Control. The controllers in situated control approaches specify how the robot should respond to sensory data. The two main representatives of the situated approach are the reactive and the behavior-based robot control.

Reactive Architectures implement robot controllers as collections of condition-action rules. These condition-action rules select the appropriate actions for each sensor reading without deliberating. Reactive control usually means two things. First, the controller has a very fast decision cycle (sense/act cycle). Second, the robot controller minimizes the use of internal state. The motto of reactive architectures is that "the world is its own best model" [38]. Having minimal state and in particular no symbolic world models avoids the efforts necessary for their construction and maintenance. Also the algorithms that use these symbolic models are often computationally very expensive. Rather than constructing and relying on world models reactive controllers rather sense their surroundings at a much higher frequency. A consequence of having stored very little information about the environment is that all possible situations have to be associated with appropriate actions. This approach works only well if the right action can always be selected almost solely on the sensor data.

Robots with reactive control systems (Pengi [2], Allen [35], Herbert [50]) work robustly in dynamic and unknown environments because they respond fast to sensor input. Disadvantages of reactive architectures are that they

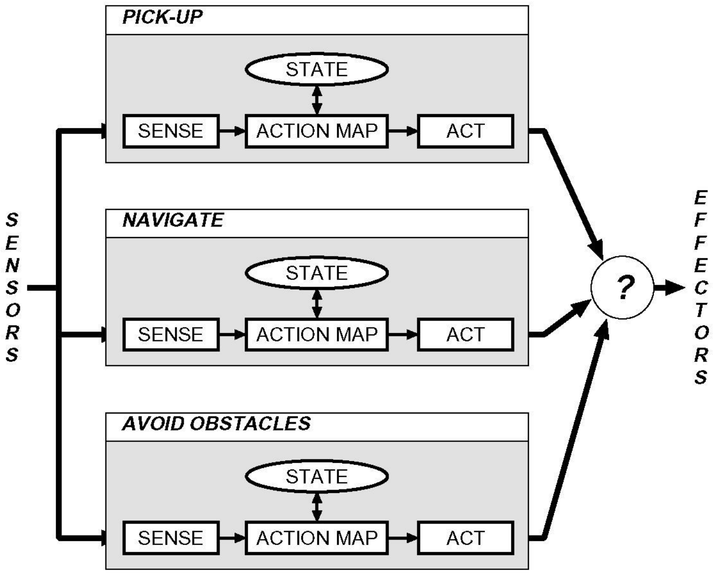


Fig. 8.4. Behavior-based robot control architecture.

can usually handle only a small set of fixed jobs of limited complexity. Also the information that can be sensed at any time has to be sufficient for selecting the right actions. While reactive approaches are characterized by fast response time to environmental events they are unable to deal with multiple and varying goals in a sophisticated manner.

Behavior-based Architectures [35, 118, 154]. Behavior-based architectures decompose the problem of autonomous control into special-purpose task-achieving modules that are connected directly to sensors and effectors. Thus, the behavior of the robot at any given moment is the result of a set of interacting continuous processes acting on input from the environment. For instance, when picking up an object the robot has to track the object and the hand in order to grasp the object. The control processes verify the results of the actions they execute by sensing the effects they have on the environment. These control processes can be activated or deactivated by asynchronous exogenous events.

Behavior-based controllers are robust against failures: modules can misbehave and sensors report incorrect information and the situation is still not dangerous for the robot. This is the case because each control process needs access only to a subset of sensors and vital behaviors overwrite control decisions of less essential behaviors. Behavior-based systems have a fast sense/act cycle. The difficult problem is task arbitration, i.e., the resolution of conflicts when multiple control processes try to control the same effector simultaneously in different ways.

Subsumption architecture [35]. The subsumption architecture is the most prominent architecture for reactive and behavior-based robot control. The main characteristics of the architecture are distributed, reflexive control and no explicit representation. The control system is organized in concurrent layers that interact (see figure 8.4). The lower levels are simple, vital behaviors like avoiding obstacles or running away; the higher levels are more complex and task-directed, like map learning. Robot controllers that are organized in a subsumption architecture can be specified in the BEHAVIOR LANGUAGE [36] and are compiled into networks of augmented finite state machines. The subsumption approach is targeted to equip autonomous robots with the behavioral repertoire typical for insects.

The COG project [40] is the biggest application of reactive and behavior-based control methods. The goal of the project is to build an integrated physical humanoid robot. Other approaches to the implementation of reactive and behavior-based control systems include situated automata [141, 143], the ALFA language [75], Situated Control Rules [62].

Structured reactive controllers add to behavior-based systems the capability to make control decisions that require foresight and a change in the intended course of action.

Hybrid Deliberative Robot Control. Hybrid architectures [93] integrate deliberation and action. They expand and schedule tasks before executing them. We will differentiate here between three types of hybrid control systems: hybrid layered control systems, bounded rationality architectures, and Belief/Desire/Intention architectures.

Hybrid layered control architectures. One approach for achieving robust execution and manageable planning problems is the use of two different languages: a structured programming language for the low-level reactive control and a very simple high-level language for strategic planning. This way the planner can still have the illusion of plans being sequences of plan steps and many existing planning techniques carry over to robot control more or less the way they are. The RAP interpreter [70, 71] is a plan interpreter that allows a planner to keep this illusion. The planner can generate partially ordered sets of actions (goal steps) and pass them to the RAP interpreter. The RAP interpreter takes these plan steps, selects methods for carrying them out based on sensor data, and executes them in a robust manner.

Typically, hybrid layered approaches have three different layers of control. First, the reactive layer that makes sure that the robot operates continuously and fast to events in the environment. Second, the planning layer that makes more strategic decisions on the course of action. Third, the sequencing layer that decouples the reactive and planning layer by maintaining a queue of actions to be taken. The actions in the queue have priorities and ordering constraints. Both, the reactive and planning layer can add actions to the queue. The planning layer can also implement ordering constraints on the queue. Timely action is guaranteed because important reactions have high priority.

The 3T [31], Animate Agent Architecture [74], and Atlantis [76] are hybrid layered control architectures that that are based on rap system. Other hybrid layered systems include [11] and SSS [51].

Structured reactive controllers can avoid flaws in the behavior of the robot that hybrid layered controllers cannot. The kinds of flaws are those that require the planner to reason about the process of plan interpretation and its effects on the behavior of the robot and those that require changes in the course of action that are neither the addition of steps nor ordering constraints.

The Reactor-Planner Architecture. [115] The kitting robot uses a control system that consists of two concurrent processes: the planner and the reactor. The planner perceives aspects of the performance of the reactor and adapts the reactors incrementally to increase the reactor's reliability of achieving a given set of goals. The reactor is written in $\mathcal{RS}$, a concurrent robot control language, which has an expressiveness similar to rpl. The main difference between Lyons' reactor-planner architecture and structured reactive controllers is that Lyons' control architecture is tailored towards highly repetitive jobs, while structured reactive controllers are designed for non-recurring sets of routine jobs.

bdi *Architectures.* [139, 140] The basic concepts in bdi architectures are beliefs, goals, intentions, actions and the relationships between them. bdi architectures can be formalized using a branching-time possible-worlds logic, where each world has a branching time future and a single past. Situations are particular time points in particular worlds and transformed into each other by events. For each situation we can specify the worlds that the agent believes to be possible, desires to bring about, and commits to achieving. The agent achieves its goals by choosing actions that bring it from the belief worlds to the intention worlds. bdi architectures represent incomplete knowledge as multiple possible worlds and alternative actions as branches within a possible world.

prs (Procedural Reasoning System) is an agent control system based on a bdi architecture [77, 78]. prs decides on actions and their order based on a database of current beliefs, a set of goals (desires), a set of intentions (goals the planner is committed to) and a set of plans specifying how to achieve goals and how to react to the environment. The prs interpreter selects appropriate plans based on the set of goals and current beliefs, adds the plans to the set of intentions and executes them.

prs and structured reactive controllers explicitly represent goals and plans and reason about them. The difference between both architectures is that prs does not make a commitment to planning techniques or the interaction between planning and execution. It leaves these aspects to the programmer to implement them when needed as meta-knowledge areas.

Bounded Rationality Architectures. A bounded optimal agent behaves as well as possible given its computational resources. Bounded optimality specifies optimal programs rather than optimal actions or optimal computation sequences. RALPH-MEA "An architecture for bounded rationality" [144] runs in parallel different methods for selecting the best actions. The methods, decision-theoretic, goal-based, action utility, and condition-action, require different types of information and time resources. An arbitrator chooses which method to use for determining the next action based on the constraints in the environment and the information available. IRMA [32] is another architecture for controlling a resource-bounded autonomous agent. It uses a filtering process to avoid wasting computational resources.

TCA. (Task Control Architecture) [150] is a specialized real-time operating system for managing the task-specific processes of a robot controller. The purpose of TCA is the handling of multiple, complex tasks in dynamic and uncertain environments by combining reactive and planning behaviors. The task-specific communicate via a central control module. TCA provides tools for interleaving planning and execution, monitoring environment changes and reacting to them, recovering from execution failures, and coordinating multiple processes.

8.4.2 Control Languages for Reactive Control

A robot control language provides the control and data abstractions that support the implementation of autonomous robot controllers (see [115] for a good discussion). In the following we will briefly describe the characteristics of different robot control languages. We will start with the L system which is a dialect of Common Lisp and can be used to implement higher-level robot control languages, in particular using Lisp's macro facility. The remaining languages are more closely tight to specific robot control architectures.

L System. [39] The L system is a development system for a highly efficient dialect of Common Lisp which includes multi-threading extensions. The L system provides a series of L functions and macros which simplify the creation of large numbers of threads which can communicate via a message passing paradigm. It also provides a real time operating system.

Behavior Language. [36] The BEHAVIOR LANGUAGE is a rule-based real-time parallel robot programming language. It compiles into a an extended version of the subsumption architecture (or Augmented Finite State Machines) [35]. Behaviors are groups of rules. All rules are assumed to run in parallel and asynchronously. ALFA [76] is another language for the implementation of subsumption architectures.

GAPPS and REX. [101] GAPPS allows for the specification of reactive plans in terms of declarative tasks. Given declarative tasks (goals) of the form do(?a), achieve(?g), or maintain(?p), and a set of goal reduction rules, the system compiles the tasks into a synchronous digital circuit that achieves the tasks. Goals and plans in GAPPS are implicit in the compiled circuit. The circuits are specified in REX [100].

Rather than synthesizing executable plans from abstract specifications XFRM improves plans during their execution and therefore has to represent goals and reactive plans in an explicit way. These explicit representations allow XFRM to revise and specialize reactive plans if the robot acquires more information about the environment or if it changes its preferences, for instance, when it changes from a slow and robust execution mode to a fast and less reliable one.

RAP. (Reactive Action Packages) [70, 73] RAP is a language for specifying situation-driven interpretation of “sketchy” plans, plans that contain steps that are not immediately executable or underspecified. The RAP system uses reactive action packages to perform sketchy plan steps. Reactive action packages contain situation-specific methods for achieving a particular goal. They apply their methods repeatedly until the goal of the plan step is achieved. The choice of the method to apply depends on the context in which the step is executed.

The RAP language is a predecessor of RPL, the robot control language used in this book. We have shown on page 37 how reactive action packages can be implemented in RPL. RPL provides a larger set of control structures and more tools for planning.

8.4.3 Robot Planning

Robot planning is the automatic generation, debugging, and optimization of robot plans, the parts of the robot controllers that planning systems reason about and manipulate [126]. Planning systems for autonomous robots acting in partly unknown and changing environments have to apply techniques to

– deal explicitly with the uncertainty that arises from the unpredictability of, and the incomplete information regarding, the environment;
– revise plans to avoid flaws in the robot’s behavior;
– decide, based on expectations, how the robot is to react to sensory input; and
– manage their computational resources.

The following sections will contrast our work with other research in these areas.

Refinement Planning. The basic idea of refinement planning is to start with an abstract plan and refine it until the plan is detailed enough to be executed by the plan interpreter. One kind of refinement planning is classical AI planning [7, 45, 96, 119], which seeks to solve the following planning problem:

> *Given a conjunctive goal G, transform the abstract plan* (ACHIEVE *G*) *into a sequence of actions S such that G is true when S is executed in the current situation.*

As important as the problem statement is what the planning system assumes about the robot and its environment. The robot is assumed to be omniscient and have perfect control over its effectors. The planning system also assumes that any change in the environment is caused by the robot's actions. The robot has a set of basic actions that can be executed one at a time.

Acting in a perfectly known and static world allows the robot to make all decisions regarding its course of action in advance. The resulting plan needs no control structures other than ordering constraints between robot actions: plans can be represented simply as a partially ordered set of actions. The assumptions also imply that plans can fail to achieve a state G only for two reasons: (1) the plan does not contain an action that causes G to hold and (2) the plan contains an action achieving G but G is then destroyed by a subsequent action. Consequently, planning systems need only two planning operations: (1) the insertion of new actions and (2) the addition of constraints on the order in which actions are to be executed. The simplicity of the plan representation, the small number of failure types and plan modification operations, and the strong assumptions allow the robot to apply simple planning algorithms that compute provably correct plans.

Unfortunately, most autonomous robots violate these assumptions. The robots have to sense their environments with noisy and restricted sensors. They robot also have to store and maintain descriptions of objects they have seen which might be incomplete and faulty.[3]

Planning with Uncertainty. Several variations of the classical planning problem have been proposed to reason about exogenous events; sensing actions, and uncertain and incomplete information about the current state of the world. For example,

Given: A conjunctive goal G
Do: transform (ACHIEVE G) into a sequence of plan steps S

1. that maximizes the probability that G is true [107, 61, 64, 55, 29, 90, 63]
2. that maximizes the expected utility [83, 81, 33, 161, 110, 163, 67]

when S is executed in the current situation.

XFRM does not reason about the exact probabilities of outcomes of plan execution. Probability is handled implicitly by the projector that generates

[3] Maes [117]: *"Although classical AI planning systems seemed to be very appealing, only very few have been used to control artificial agents in realistic domains and those that have been used exhibited serious limitations with respect to their real-time problem-solving ability, flexibility in execution, and robustness".*

random execution scenarios based on the given probability distributions. The planning approaches that maximize the probability of goal achievement only work if the plans are sequences of discrete plan steps and probability distributions describing the current state of the world are available. XFRM, on the other hand, reasons about plans that specify the concurrent execution of continuous control processes and has only sparse information about the current state of the world. XFRM's weaker methods suffice because the plans it reasons about are more robust and cause similar behavior in a range of standard situations. Another advantage of XFRM's Monte Carlo projection method is that it is much faster than other methods for probabilistic temporal projection [165].

The utility models used by XFRM describe the quality of structured reactive plans in terms of their robustness (probability of complete goal satisfaction), the estimated execution time, and their completeness and are a limited model of plan utility. We have restricted ourselves to these aspects, because so far these are the ones that can be checked by the plan simulator. A thorough discussion of more expressive utility models for plans, including resources other than time, partial goal satisfaction, etc., can be found in [81].

Transformational Planning. Transformational planning is a planning technique which modifies a single plan instance instead of refining an abstract plan. The transformational planners that are most closely related to XFRM are HACKER [156, 155], CHEF [85, 86], and GORDIUS [148, 151, 149].

Given a set of tasks, transformational planners generate a plan hypothesis that can be projected and revise the plan hypothesis until it is good enough. Transformational planners assume that it is more efficient to compute an almost correct plan and debug it later than trying to compute a correct plan in the first place. Thus instead of constructing plans from primitive actions over and over again, a planner retrieves large parts of plans and combines them using heuristic methods. The transformational planning approach is very promising in applications where we have good heuristics that can generate "almost correct" plans, methods for finding all flaws in a plans, and good knowledge about how to debug plans. Sussman [155] was the first one who suggested that constructing almost correct solutions and debugging them later is a powerful problem solving strategy.

Once a plan failure is detected the planners have to find out why and where in order to revise the plan appropriately. This step is implemented in GORDIUS [149] as the localization of faulty assumptions made by the planner or projector. In the CHEF planner a detailed explanation for bugs is generated by executing questioning scripts [86]. The transformations for plan repair are indexed either by the faulty assumptions (GORDIUS) or by explanations (CHEF).

The main difference between XFRM and other transformational planners is that XFRM reasons about concurrent robot control programs while CHEF and GORDIUS reason about plans that are sequences of plan steps, and HACKER about plans that contain steps, loops and conditionals. Another difference is that HACKER, GORDIUS, and CHEF try to produce correct plans, while XFRM adapts plans to specific situations to avoid particular flaws.

Planning Reactive Behavior. Robot planners have to synthesize plans that implement any problem-solving behavior necessary to solve complex tasks in its environment — not just sequences of primitive robot actions [123].

ERE (Entropy Reduction Engine) [63] contains a synthetic projection algorithm. Given given a behavioral constraint and a probabilistic causal theory of the robot actions, ERE attempts to find a directed acyclic graph of situation control rules [62] that maximizes the probability of successful task performance. The planning algorithm consists of two steps: *traverse* and *robustify*. The *traverse* step constructs a sequence of steps that achieves the given goals efficiently and with relatively high probability. The *robustify* step determines most likely deviations and call traverse for them in order to find ways to recover from them. The ERE planning algorithm makes the plan incrementally more robust. ERE and XFRM share a number of similarities: they use probabilistic anytime algorithms for projecting plans and improving their robustness. In contrast, XFRM computes plans that are represented in RPL, a more complex plan representation language.

Dean et al. [55] propose a planning algorithm with the same purpose as ERE based on the theory of Markov decision problems. Another successor of the ERE planning algorithm is the reaction-first algorithm [64] that computes plan prefixes that can be substituted for the default plan to increase the probability of goal satisfaction.

PROPEL [113] is a planning system that reasons about plans written in a non-deterministic plan language with loops, conditionals, and a CHOOSE-operator for variable instantiations and selecting procedures to be executed. PROPEL starts with a plan that contains choices that are to be made by simple heuristics. The job of the planner is to make more informed choices based on its projected expectations about the effects of each alternative. The difference between PROPEL and XFRM is that PROPEL replaces choice points in a plan by more informed decisions while XFRM revises a course of action to avoid flawed behavior.

ROBOSOAR [108] controls a robot with a set of production rules that map sensor data into effector commands. In routine situations exactly one rule fires; in exceptional states multiple rules may be applicable. To resolve such impasses, the planner is called and chooses the best rule to apply. After the impasse is resolved, ROBOSOAR modifies the rule set controlling the robot such that the right rule will fire the next time without causing an impasse. The purpose of XFRM is not to learn how to behave in particular situations but to adapt the behavior of the robot to exceptional situations. Since those exceptional situations are assumed to occur rarely, XFRM does not change its routine plans to accommodate these additional situations.

Time-dependent Planning. A robot planning system has to treat planning time as a limited resource; i.e., it has to return plans for any allocation of computation time and has to reason about whether the expected gains from further planning will outweigh the costs of spending more planning time [53, 30, 89]. Anytime algorithms are algorithms that return answers for any allocation of computation time, and are expected to return better answers when more time is given (for example, iterative deepening [106] is an anytime search algorithm) The reasoning technique which copes with problems of assigning time resources to subproblems in a problem-solving process is called *deliberation scheduling* [30, 99]. The goal of deliberation scheduling is to find an assignment of time resources to computation problems and planning efforts such that running the algorithms as specified in the schedule will produce a plan with maximal expected quality.

The XFRM planning algorithm is an anytime algorithm but so far we have not incorporated any sophisticated methods for deliberation scheduling. In chapter 7 we have described the basic mechanisms of structured reactive controllers for controlling planning processes but the problem of reasoning about when to plan what and for how long has not been addressed (but see [167, 145, 146] for work in this area).

9. Conclusion

This book is part of an investigation that explores whether planning the behavior of an autonomous robot in a changing and partly unknown world is possible and can improve the robot's performance. The answer is yes — if the robot control systems are realized as structured reactive controllers.

Structured reactive controllers are collections of concurrent control routines that specify routine activities and can adapt themselves to non-standard situations by means of planning. They will outperform robots with fixed controllers, if most but not all jobs can be performed by flexible routine activity and the behavior of the robot in non-standard situations is predictable. Many service robots, like delivery robots, share these problem-solving characteristics.

We have conducted experiments in a simulated changing world using a simulated robot with limited and noisy sensing and imperfect control. These experiments have shown that structured reactive controllers are in situations that do not require foresight as efficient as fixed robot controllers and in situations that require foresight more reliable than their fixed counterparts. Other experiments have demonstrated that controllers that forestall flaws in structured reactive plans act competently in situations that neither fixed nor by deliberative robot controllers can handle.

9.1 What Do Structured Reactive Controllers Do?

Structured reactive controllers are means for controlling autonomous robots that perform routine jobs in changing and partly unknown environments. They enable robots to act flexibly and reliably in standard situations and anticipate and forestall flaws in their activities. Forestalling predictable behavior flaws gives the robots the competency to cope more flexibly with the situations they encounter and better utilize the information they acquire while carrying out their jobs. For example, when a robot detects another robot, structured reactive controllers predict that robot's effect on its plan and — if necessary — revise its plan to make it more robust.

Structured reactive controllers work as follows. When given a set of jobs, the structured reactive controller retrieves default routines for individual jobs and executes the routines concurrently. These routine activities are general and flexible — they work for standard situations and when executed concurrently with other routine activities. Routine activities can cope well with partly unknown and changing environments, run concurrently, handle interrupts, and control robots without assistance over extended periods. For standard situations, the execution of these routine activities causes the robot to exhibit an appropriate behavior in achieving their purpose. While it executes routine activities, the robot controller also tries to determine whether its routines might interfere with each other and watches out for non-standard situations. If it encounters a non-standard situation it will try to anticipate and forestall behavior flaws by predicting how its routine activities might work in the non-standard situation and, if necessary, revising its routines to make them robust for this kind of situation. Finally, it integrates the proposed revisions smoothly into its ongoing course of actions.

9.2 Why Do Structured Reactive Controllers Work?

The following ones are the most important reasons that make the success of structured reactive controllers possible.

1. Structured reactive plans specify flexible routine activity, which is the result of concurrent sensor-driven behaviors. Routine activity can cope well with partly unknown and changing environments, handles interruptions, and controls robots without assistance over extended periods. Structured reactive controllers revise their course of action in ways other planning systems cannot. A revision like "do as much of a particular job as you can until a given deadline passes, and do the rest of the job after all other jobs are completed" cannot be performed by systems that reason about plans that lack control abstractions for interrupting plan steps and resuming them later. Structured reactive controllers can forestall flaws in a course of action without requiring the robot to stop its current activity. Only plan languages that allow for sophisticated synchronization of concurrent control processes, monitoring of the execution, adequate reaction to unpredicted events, and recovery from local execution failures make it possible to
   - specify flexible interruptable routine activity,
   - change the robot's course of action in non-trivial ways, and
   - revise plans smoothly during their execution,
2. A small number of randomly projected execution scenarios per candidate plan suffices to improve the average performance of a robot by anticipating and forestalling behavior flaws. Thus, a Monte Carlo projection algorithms can be used to make predictions about the temporally complex and probabilistic behavior caused by structured reactive controllers. A small number of projected execution scenarios are enough because (1)

routine plans are designed such that they are unlikely to cause behavior flaws in standard situations; and (2) if a routine plan is likely to cause a behavior flaw in situations that satisfy a condition *c* and the robot has detected *c* then the behavior flaws will probably be projected.

3. Structured reactive controllers forestall a large variety of typical flaws in autonomous robot behavior. I have described a taxonomy of behavior flaws that contained models of more than twenty general categories of behavior flaws. The taxonomy covers behavior flaws of the delivery robot that might be caused by sensing routines that overlook or misperceive objects, by unfaithful tracking of objects through their descriptions, by unreliable physical actions with undesired effects, or by interference between multiple threads of the robot's own plan. Most of these behavior flaws that are common in the execution of robot control programs cannot be handled by other planning systems.
   Structured reactive controllers can predict and forestall such flaws because their planning processes infer whether particular parts of the plan were executed and why. The use of XFRML makes such inferences possible because XFRML not only represents the physical effects of plan execution, but also the process of plan interpretation, as well as temporal, causal, and teleological relationships among plan interpretation, the world, and the physical behavior of the robot.
4. Structured reactive controllers use fast algorithms for the construction of default plans, the diagnosis of behavior flaws, and editing subplans during there execution. To allow for fast algorithms structured reactive plans are designed to be *transparent*, *restartable*, and *coherent*. Transparency enables the planner to understand important aspects of the plan by syntactic matching and pattern-directed retrieval. Restartability enables the robot to integrate a plan revision into a subplan by simply repeating the execution of the subplan. Coherence simplifies the construction of routine plans because coherent plans can run concurrently without additional synchronizations and still achieve a coherent problem-solving behavior.
   In particular, transparent plans make the following computations simple and efficient: infering the purpose of a subplan, finding a subplan with a given purpose, automatically generating a plan that can accomplish a given purpose, determining flaws in the behavior that is caused by a subplan, and estimating how good the behavior caused by a subplan is with respect to the robot's utility model. For this purpose, I have extended RPL with declarative statements to make structured reactive plans transparent.
   We can make sure that the planning processes only reason about plans that have these properties by designing the plan schemata in the plan library such that they satisfy these properties and the construction of the default plan and the application of plan revision methods such that they preserve the properties. This can be done without restricting the expressiveness of the plan language.

9.3 Do Structured Reactive Controllers Work for Real Robots?

To validate that structured reactive controllers can perform reliable and effective control for autonomous mobile robots, we have carried out two kinds of experiments. The first one is a long term experiment in which we have implemented a structured reactive controller as a high-level controller for an interactive museums tourguide robot. The tourguide robot, called MINERVA, has operated for a period of thirteen days in the Smithsonian's National Museum of American History [159]. In this period, it has been in service for more than ninetyfour hours, completed 620 tours, showed 2668 exhibits, and travelled over a distance of more than fortyfour kilometers. RPL controlled MINERVA's course of action in a feedback loop that was carried out more than three times a second. MINERVA used plan revision policies for the installment of new commands, the deletion of completed plans, and tour scheduling and performed about 3200 plan adaptations. The MINERVA experiment demonstrates that structured reactive controllers can (1) reliably control an autonomous robot over extended periods of time and (2) reliably revise plans during their execution.

Plan revision techniques which are capable of improving the average performance of the robot were tested in a second category of experiments in which we ran isolated problem-solving scenarios to demonstrate specific features of structured reactive controllers, such as the integration of scheduling capabilities [16] or the prediction capabilities [18].

References

1. P. Agre. The dynamic structure of everyday life. Technical Report AI-TR-1087, AI lab, MIT, October 1988. Could be AI-TR-1085.
2. P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, Seattle, WA, 1987.
3. P. Agre and I. Horswill. Cultural support for improvisation. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 363–368, San Jose, CA, 1992.
4. P. Agre and S. Rosenschein, editors. *Computational Theories of Interaction and Agency.* The MIT Press, Cambridge, MA, 1996.
5. P. E. Agre and D. Chapman. What are plans for? In P. Maes, editor, *Designing Autonomous Agents*, pages 17–34. MIT Press, Cambridge, MA, 1990.
6. J. Allen and G. Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, October 1994.
7. J. Allen, J. Hendler, and A. Tate. *Readings in Planning.* Representation and Reasoning Series. Morgan Kaufmann, San Mateo, California, 1990.
8. J. Aloimonos, A. Bandopadhay, and I. Weiss. Active vision. *International Journal of Computer Vision*, 1(4):333–356, 1988.
9. J. Ambros-Ingerson and S. Steel. Integrating planning, execution, and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 735–740, St. Paul, MN, 1988. Morgan Kaufmann.
10. T. Arbuckle and M. Beetz. Extensible, runtime-configurable image processing on robots — the recipe system. In *Proceedings of the 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1999.
11. R. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In S. S. Iyengar and A. Elfes, editors, *Autonomous Mobile Robots: Control, Planning, and Architecture (Vol. 2)*, pages 162–169. IEEE Computer Society Press, Los Alamitos, CA, 1991.
12. R. Bajcsy. Active perception. *Proceedings of the IEEE*, 76(8):996–1005, August 1988.
13. D. Ballard. Animate vision. *Artificial Intelligence*, 48:57–86, 1991.
14. M. Beetz. Structured reactive controllers for service robots in human working environments. In G. Kraetzschmar and G. Palm, editors, *Hybrid Information Processing in Adaptive Autonomous Vehicles.* Springer, 1998. to appear.
15. M. Beetz, T. Arbuckle, A. Cremers, and M. Mann. Transparent, flexible, and resource-adaptive image processing for autonomous service robots. In H. Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 632–636, 1998.
16. M. Beetz and M. Bennewitz. Planning, scheduling, and plan execution for autonomous robot office couriers. In R. Bergmann and A. Kott, editors, *Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments, Workshop Notes 98-02.* AAAI Press, 1998.

17. M. Beetz, W. Burgard, D. Fox, and A. Cremers. Integrating active localization into high-level control systems. *Robotics and Autonomous Systems*, 23:205–220, 1998.
18. M. Beetz and H. Grosskreutz. Causal models of mobile service robot behavior. In *Fourth International Conference on AI Planning Systems*, page 163, 1998.
19. M. Beetz and D. McDermott. Declarative goals in reactive plans. In J. Hendler, editor, *Proceedings of the First International Conference on AI Planning Systems*, pages 3–12, Washington, DC, 1992. Morgan Kaufmann.
20. M. Beetz and D. McDermott. Improving robot plans during their execution. In Kris Hammond, editor, *Second International Conference on AI Planning Systems*, pages 3–12, Chicago, IL, 1994. Morgan Kaufmann.
21. M. Beetz and D. McDermott. Improving robot plans during their execution. In Kris Hammond, editor, *Proceedings of the Second International Conference on AI Planning Systems*, pages 3–12, Chicago, IL, 1994. Morgan Kaufmann.
22. M. Beetz and D. McDermott. Executing structured reactive plans. In Louise Pryor and Sam Steel, editors, *AAAI Fall Symposium: Issues in Plan Execution*, 1996.
23. M. Beetz and D. McDermott. Local planning of ongoing activities. In Brian Drabble, editor, *Third International Conference on AI Planning Systems*, pages 19–26, Edinburgh, GB, 1996. Morgan Kaufmann.
24. M. Beetz and D. McDermott. Expressing transformations of structured reactive plans. In *Recent Advances in AI Planning. Proceedings of the 1997 European Conference on Planning*, pages 64–76. Springer Publishers, 1997.
25. M. Beetz and D. McDermott. Fast probabilistic plan debugging. In *Recent Advances in AI Planning. Proceedings of the 1997 European Conference on Planning*, pages 77–90. Springer Publishers, 1997.
26. M. Beetz and H. Peters. Structured reactive communication plans — integrating conversational actions into high-level robot control systems. In *Proceedings of the 22nd German Conference on Artificial Intelligence (KI 98), Bremen, Germany.* Springer Verlag, 1998.
27. Michael Beetz, Markus Giesenschlag, Roman Englert, Eberhard Gülch, and A. B. Cremers. Semi-automatic acquisition of symbolically-annotated 3d-models of office environments. In *Proc. Int'l Conf. on Robotics and Automation*, 1998.
28. S. Biundo, D. Dengler, and J. Köhler. Deductive planning and plan reuse in a command language environment. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 628–632, Vienna, Austria, August 1992. John Wiley & Sons.
29. J. Blythe. AI planning in dynamic, uncertain domains. In *Extending Theories of Action: Formal Theory & Practical Applications: Papers from the 1995 AAAI Spring Symposium*, pages 28–32. AAAI Press, Menlo Park, California, 1995.
30. M. Boddy and T. Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 979–984, Detroit, MI, 1989.
31. P. Bonasso, H. Antonisse, and M. Slack. A reactive robot system for find and fetch tasks in an outdoor environment. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, 1992.
32. M. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge, Massachusetts, 1987.
33. J. Breese and M. Fehling. Decision-theoretic control of problem solving: Principles and architecture. In *Proceedings of the 1988 Workshop on Uncertainty and Artificial Intelligence*, 1988.

34. R. Brooks. Achieving Artificial Intelligence through building robots. AI Memo 899, MIT AI Lab, Cambridge, MA, May 1986.
35. R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23, 1986.
36. R. Brooks. The behavior language; user's guide. AI Memo 1227, MIT AI Lab, Cambridge, MA, 1990.
37. R. Brooks. Intelligence without reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 569–595, Sidney, Australia, 1991.
38. R. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.
39. R. Brooks. L: A subset of common lisp. Technical report, MIT AI Lab, 1993.
40. R. Brooks and L. Stein. Building brains for bodies. *Autonomous Robots*, 1:7–25, 1994.
41. M. Broy and P. Pepper. Program development as a formal activity. In C. Rich and R. C. Waters, editors, *Artificial Intelligence and Software Engineering*, pages 123–131. 1986.
42. W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, FL, 1998.
43. T. Bylander. Complexity results for planning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 274–279, Sidney, Australia, 1991.
44. J. Carbonell, J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Kahn, C. Knoblock, S. Minton, A. Perez, S. Reilly, M. Veloso, and X. Wang. Prodigy 4.0: the manual and tutorial. Technical Report, Carnegie Mellon, 1992.
45. D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.
46. D. Chapman. *Vision, Instruction, and Action.* The MIT Press, Cambridge, MA, 1991.
47. P. Cohen, M. Greenberg, D. Hart, and A. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, pages 32–48, 1989.
48. P. Cohen and A. Howe. How evaluation guides ai research. *AI Magazine*, pages 35–44, 1988.
49. J. Connell. Creature design with the subsumption architecture. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1124–1126, Milan, Italy, 1987.
50. J. Connell. *Minimalist Mobile Robotics: A Colony-style Architecture for a Mobile Robot.* Academic Press, Boston, MA, 1990.
51. J. Connell. SSS: A hybrid architecture applied to robot navigation. In *Proceedings IEEE International Conference on Robotics and Automation*, 1992.
52. E. Davis. Semantics for tasks that can be interrupted or abandoned. In J. Hendler, editor, *Proceedings of the First International Conference on AI Planning Systems*, pages 37–44, Washington, DC, 1992.
53. T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, St. Paul, MN, 1988.
54. T. Dean and P. Bonasso. The 1992 AAAI robot exhibition and competition. *AI Magazine*, 14(1):35–48, 1993.

55. T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 574–579, Washington, DC, 1993.
56. T. Dean and M. Wellmann. *Planning and Control.* Morgan Kaufmann Publishers, San Mateo, CA, 1991.
57. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, 32(3):505–536, July 1985.
58. G. DeJong and S. Bennett. Extending classical planning to real-world execution with machine learning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1153–1161, Montreal, Canada, 1995.
59. B. Donald. Planning multi-step error detection and recovery strategies. *The International Journal of Robotics Research*, 9(1):3–60, 1990.
60. R. Doyle, D. Atkinson, and R. Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 268–272, Philadelphia, PA, 1986.
61. D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In K. Hammond, editor, *Proceedings of the Second International Conference on AI Planning Systems*, Chicago, IL, 1994. Morgan Kaufmann.
62. M. Drummond. Situated control rules. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *KR'89: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–113. Kaufmann, San Mateo, CA, 1989.
63. M. Drummond and J. Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 138–144, Boston, MA, 1990.
64. M. Drummond, K. Swanson, J. Bresina, and R. Levinson. Reaction-first search. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1993.
65. S. Engelson. *Passive Map Learning and Visual Place Recognition.* Technical report 1032, Yale University, Department of Computer Science, May 1994.
66. K. Erol, D. S. Nau, and V. S. Subrahmanian. On the complexity of domain-independent planning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 381–386, San Jose, CA, 1992.
67. O. Etzioni. Tractable decision-analytic control. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR89)*, pages 114–125. Kaufmann, San Mateo, CA, 1989.
68. R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):189–208, 1972.
69. R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
70. J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* Technical report 672, Yale University, Department of Computer Science, 1989.
71. J. Firby. Building symbolic primitives with continuous control routines. In J. Hendler, editor, *Proceedings of the First International Conference on AI Planning Systems*, pages 62–69, Washington, DC, 1992. Morgan Kaufmann.
72. J. Firby. Task networks for controlling continuous processes. In Kris Hammond, editor, *Proceedings of the Second International Conference on AI Planning Systems*, pages 49–54, Chicago, IL, 1994. Morgan Kaufmann.
73. J. Firby. The RAP language manual. Animate Agent Project Working Note AAP-6, University of Chicago, 1995.

74. J. Firby, R. Kahn, P. Prokopowitz, and M. Swain. An architecture for vision and action. In C. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 72–79, Montreal, Canada, 1995.
75. E. Gat. Alfa: A language for programming reactive robotic control systems. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 1116–1121, 1991.
76. E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, 1992.
77. M. Georgeff and F. Ingrand. Decision making in an embedded reasing system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 972–978, Detroit, MI, 1989.
78. M. Georgeff and A. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682, Seattle, WA, 1987.
79. G. Giralt, R. Sobek, and R. Chatila. A multi-level planning and navigation system for a mobile robot: A first approach to hilare. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 335–337, Tokio, 1979.
80. R. Goodwin. Reasoning about when to start acting. In Kris Hammond, editor, *Proceedings of the Second International Conference on AI Planning Systems*, pages 86–91, Chicago, IL, 1994. Morgan Kaufmann.
81. P. Haddawy and S. Hanks. Issues in decision-theoretic planning: Symbolic goals and utilities. In K. Sycara, editor, *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 48–58, 1990.
82. P. Haddawy and S. Hanks. Representations for decision-theoretic planning: Utility functions for deadline goals. In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 71–82, Cambridge, MA, October 1992. Morgan Kaufmann.
83. P. Haddawy and L. Rendell. Planning and decision theory. *The Knowledge Engineering Review*, 5:15–33, 1990.
84. B. Hallam and G. Hayes. Comparing robot and animal behavior. DAI Research Report 598, University of Edinburgh, 1994.
85. K. Hammond. *Case-Based Planning.* Academic Press, Inc., 1989.
86. K. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45(1):173–228, 1990.
87. K. Hammond, T. Converse, and J. Grass. The stabilization of environments. *Artificial Intelligence*, 73, 1995.
88. W. Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51:223–271, 1991.
89. S. Hanks. Controlling inference in planning systems: Who, what, when, why, and how. Technical Report #90-04-01, Department of Computer Science, University of Washington, 1990.
90. S. Hanks. Practical temporal projection. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 158–163, Boston, MA, 1990.
91. S. Hanks. *Projecting Plans for Uncertain Worlds.* Technical report 756, Yale University, Department of Computer Science, 1990.
92. S. Hanks and A. Badr. Critiquing the TILEWORLD: Agent architectures, planning benchmarks, and experimental methodology. Technical Report 91-10-31, Dept. of Computer Science and Engineering, Univ. of Washington, 1991.

93. S. Hanks and R. J. Firby. Issues and architectures for planning and execution. In *Proceedings of the Workshop on Innovative Approaches to Planning*, pages 59–70, Scheduling and Control, San Diego, CA, 1990.
94. S. Hanks, M. Pollack, and P. Cohen. Benchmarks, test beds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 14(4):17–42, 1993.
95. B. Hayes-Roth. Intelligent monitoring and control. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 243–249, Detroit, MI, 1989.
96. J. Hendler, A. Tate, and M. Drummond. AI planning: Systems and techniques. *AI Magazine*, 11(3):61–77, 1990.
97. I. Horswill. Polly: A vision-based artificial agent. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC, 1993.
98. I. Horswill. Analysis of adaptation and environment. *Artificial Intelligence*, 73:1–30, 1995.
99. E. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 111–116, St. Paul, MN, 1988.
100. L. Kaelbling. REX: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of AIAA Conference on Computers in Aerospace*, Wakefield, MA, 1987.
101. L. Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 60–65, St. Paul, MN, 1988.
102. S. Kambhampati. A theory of plan modification. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.
103. K. Kanazawa. *Reasoning about Time and Probability.* PhD thesis, Brown University, 1992.
104. S. Kannan and M. Blum. Designing programs that check their work. In *Proceedings of the 1989 Symposium on Theory of Computing*, 1989.
105. K. Konolige. Designing the 1993 robot competition. *AI Magazine*, 15(1):57–62, 1994.
106. R. Korf. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
107. N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.
108. P. Laird, E. Yager, M. Hucka, and M. Tuck. Robo-Soar: an integration of external interaction, planning, and learning in Soar. *Robotics and Autonomous Systems*, 8:113–129, 1991.
109. P. Langley, H. Simon, G. Bradshaw, and J. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Processes.* The MIT Press, Cambridge, MA, 1987.
110. C. Langlotz, L. Fagan, S. Tu, J. Williams, and B. Sikic. Onyx: An architecture for planning in uncertain environments. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 447–449, Los Angeles, CA, 1985.
111. J.-C. Latombe. *Robot Motion Planning.* Kluwer, Dordrecht, The Netherlands, 1991.
112. N. Lehrer. KRSL specification language. Technical Report Technical Report 2.0.2, ISX Corporation, 1993.
113. R. Levinson. A general programming language for unified planning and control. *Artificial Intelligence*, 76:319–375, 1995.

114. V. Lumelski and A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Transactions on Automatic Control*, 31(11):1058–1063, 1986.
115. D. Lyons and M. Arbib. A formal model of computation for sensory-based robotics. *IEEE Journal of Robotics and Automation*, 5(3):280–293, 1989.
116. D. Lyons and A. Hendriks. A practical approach to integrating reaction and deliberation. In J. Hendler, editor, *Proceedings of the First International Conference on AI Planning Systems*, pages 153–162, Washington, DC, 1992.
117. P. Maes, editor. *Designing Autonomous Agents: Theory and Practice from Biology and Back*. MIT Press, Cambridge, MA, 1990.
118. M. Mataric. Behavior-based systems: Main properties and implications. In *Proceedings IEEE International Conference on Robotics and Automation, Workshop on Intelligent Control Systems*, pages 46–54, 1992.
119. D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639, Anaheim, CA, 1991.
120. D. McDermott. Planning and acting. *Cognitive Science*, 2(2):71–109, 1978.
121. D. McDermott. Artificial intelligence meets natural stupidity. In J. Haugeland, editor, *Mind Design: Philosophy, Psychology, Artificial Intelligence*, pages 143–160. MIT Press, Cambridge, MA, 1981.
122. D. McDermott. Reasoning about plans. In J. R. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*, pages 269–317. Ablex, Norwood, NJ, 1985.
123. D. McDermott. Planning reactive behavior: A progress report. In K. Sycara, editor, *Innovative Approaches to Planning, Scheduling and Control*, pages 450–458, San Mateo, CA, 1990. Kaufmann.
124. D. McDermott. A reactive plan language. Research Report YALEU/DCS/RR-864, Yale University, 1991.
125. D. McDermott. Regression planning. *International Journal of Intelligent Systems*, 6:357–416, 1991.
126. D. McDermott. Robot planning. *AI Magazine*, 13(2):55–79, 1992.
127. D. McDermott. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University, 1992.
128. D. McDermott. An algorithm for probabilistic, totally-ordered temporal projection. Research Report YALEU/DCS/RR-1014, Yale University, 1994.
129. D. McDermott. The other problem with classical planning. Invited Talk at the AIPS-94, 1994.
130. D. McFarland and T. Boesser. *Intelligent Behavior in Animals and Robots*. MIT Press, Cambridge, MA, 1993.
131. T. Mitchell. Becoming increasingly reactive. In K. Sycara, editor, *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 459–467, 1990.
132. T. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, and J. Schlimmer. Theo: a framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, 1990.
133. B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76:427–454, 1995.
134. A. Newell. Artificial intelligence and the concept of mind. In Schank and Colby, editors, *Computer Models of Language and Thought*. 1973.
135. Allan Newell and Herbert A. Simon. GPS, A program that simulates human thought. In Edward Feigenbaum A. and Julian Feldman, editors, *Computers and Thought*, pages 279–293. McGraw-Hill, New York, 1963.
136. N. Nilsson. Shakey the robot. Technical Note 323, SRI AI Center, 1984.

137. K. Passino and P. Antsaklis. A system and control-theoretic perspective on artificial intelligence planning systems. *Applied Artificial Intelligence*, 3:1–32, 1989.
138. M. Pollack and M. Ringuette. Introducing the tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189, Boston, MA, 1990.
139. A. Rao and M. Georgeff. Modeling rational agents within a bdi-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Knowledge Representation and Reasoning (KR'91)*, pages 473–484, San Mateo, CA, 1991. Kaufmann.
140. A. Rao and M. Georgeff. An abstract architecture for rational agents. In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR'92)*, pages 439–449, San Mateo, CA, 1992. Kaufmann.
141. S. Rosenschein. Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3:345–357, 1985.
142. S. Rosenschein and L. Kaelbling. The synthesis of digital machines with provable epistemic properties. In *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 83–98, Monterey, CA, 1986.
143. S. Rosenschein and L. Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73:149–173, 1995.
144. S. Russell. An architecture for bounded rationality. *SIGART Bulletin 2*, 1991.
145. S. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 3, 1995.
146. S. Russell and E. Wefald. *Do the right thing: studies in limited rationality.* MIT Press, Cambridge, MA, 1991.
147. M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, Milan, Italy, 1987.
148. R. Simmons. *Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems.* Technical report 1048, MIT AI Lab, 1988.
149. R. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 94–99, St. Paul, MN, 1988.
150. R. Simmons. An architecture for coordinating planning, sensing, and action. In K. Sycara, editor, *Innovative Approaches to Planning, Scheduling and Control*, pages 292–297, San Mateo, CA, 1990. Kaufmann.
151. R. Simmons. The roles of associational and causal reasoning in problem solving. *Artificial Intelligence*, 53:159–207, 1992.
152. R. Simmons. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 34–43, 1994.
153. R. Simmons. The 1994 AAAI robot competition and exhibition. *AI Magazine*, 16(2):19–30, 1995.
154. L. Steels. Building agents out of behavior systems. In L. Steels and R. Brooks, editors, *The Artificial Life Route to Artificial Intelligence. Building Situated Embodied Agents.* Lawrence Erlbaum Ass., 1994.
155. G. Sussman. The virtuous nature of bugs. In *1st Conf. of the Society for AI and the Simulation of Behaviour (AISB 74)*, pages 224–237, Sussex University, Brighton, UK, 1974.
156. G. Sussman. *A Computer Model of Skill Acquisition*, volume 1 of *Aritficial Intelligence Series.* American Elsevier, New York, NY, 1977.

157. D. Terzopoulos, X. Tu, and R. Grzeszczuk. Artificial fishes: Autonomous locomotion, perception, behavior, and learning in a simulated physical world. *Artificial Life*, 1(4):327–351, 1994.
158. S. Thrun. "personal communication", 1995.
159. S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Haehnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Minerva: A second generation mobile tour-guide robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99)*, 1999.
160. S. Ullman. Visual routines. *Cognition*, 18:97–160, 1984.
161. M. Wellmann. *Formulation of Tradeoffs in Planning under Uncertainty.* Pitman and Morgan Kaufmann, London, GB, 1990.
162. D. Wilkins. *Practical Planning: Extending the AI Planning Paradigm.* Morgan Kaufmann, San Mateo, CA, 1988.
163. M. Williamson and S. Hanks. Utility-directed planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, page 1498, Seattle, WA, 1995.
164. P. Winston. The MIT robot. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 431–463. John Wiley and Sons, New York, NY, 1972.
165. E. Yampratoom. Using simulation-based projection to plan in an uncertain and temporally complex world. Technical Report 531, University of Rochester, CS Deptartment, 1994.
166. S. Zilberstein. On the utility of planning. In M. Pollack, editor, *SIGART Bulletin Special Issue on Evaluating Plans, Planners, and Planning Systems*, volume 6. ACM, 1995.
167. S. Zilberstein and S. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 79(2), 1995.