Task-level robot programming combining object-oriented design and synchronous approach : a tentative study

Eve Coste-Maniere, Bernard Espiau, Eric Rutten

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1441

Programme 4
Robotique, Image et Vision

TASK-LEVEL ROBOT PROGRAMMING COMBINING OBJECT-ORIENTED DESIGN AND SYNCHRONOUS APPROACH : A TENTATIVE STUDY

Eve COSTE-MANIÈRE
Bernard ESPIAU
Eric RUTTEN

Juin 1991

Task-Level Robot Programming Combining Object-Oriented Design and Synchronous Approach:

a Tentative Study

Programmation au niveau tâche en robotique associant une modélisation objet et une approche synchrone :

une étude préliminaire

Mai 1991

Programme 4
ROBOTIQUE, IMAGE ET VISION

Eve Coste-Manière*, Bernard Espiau*, Eric Rutten**

(*)Ecole des Mines de Paris, Sophia-Antipolis
Rue Claude Daunesse 06565 Valbonne Cedex, France
Fax (33) 93 65 43 04, e-mail espiau@carolus.cma.fr
(**)INRIA Sophia-Antipolis
2004 Route des Lucioles 06565 Valbonne Cedex, France

Abstract

This report presents a new approach to the design of elementary robot actions and to the programming of applications. It follows a description given in [Simon e.a.91]. The three basic features of the proposed methodology are:

1. the choice of the synchronous language ESTEREL as the unique way of expression, at the 'local' level as well as at the application level, in order to ensure a coherent code generation associated with a global automaton ;
2. an object-oriented design of all entities which constitute a robot-task, aimed to the further achievement of an efficient man-machine interface ;
3. an high-level application description through a plan which allows to easily specify logical and temporal relations between robot-tasks while avoiding that an end-user writes any ESTEREL code.

The first part of the report describes the used models, with focus on the object called 'robot-task'. It is indeed a kind of elementary robot action which implements a dedicated control scheme and is provided with its own ESTEREL-coded behaviour. The second part presents a prototype of application programming language based on the previous work of one of the authors, and describes how it is translated in ESTEREL. Three examples in the area of manufacturing robotics are finally examinated.

Résumé

Ce rapport décrit une approche spécifique pour la conception de tâches robots et la programmation des applications, en prolongement du schéma donné dans un rapport de recherche précédent [Simon e.a.91]. Trois hypothèses de travail ont été retenues:

1. l'utilisation du langage ESTEREL, aussi bien au niveau local (tâche robot) qu'à celui de l'application, de facon à disposer d'un code cohérent produisant un automate global ;
2. la structuration sous forme de modèle objet de tous les éléments entrant dans la constitution d'une tâche robot dans le but ultérieur de réaliser une interface homme-machine efficace ;
3. la description haut niveau d'une application sous forme d'un plan spécifiant clairement les dépendances logiques et temporelles des tâches robot sans que l'utilisateur ait cependant à écrire du code ESTEREL.

La première partie du rapport décrit les modèles utilisés et détaille l'objet "tâche robot", sorte d'action élémentaire implémentant une loi de commande et dotée d'un comportement codé en ESTEREL. La deuxième présente le langage de programmation des applications et son mode de traduction en ESTEREL. Quelques exemples, issus du domaine de la robotique manufacturière, sont présentés.

Contents

1 Introduction

1.1 Background

We consider in this research report some high level programming aspects of a robotic system, not from a quite general point of view, but in the exact continuation of the approach described in [Borrelly e.a.90], [Simon e.a.91], [Samson e.a.91], [Coste-Manière89]. While strongly inviting the reader to consult these references, let us very briefly recall the main associated issues as stated in [Simon e.a.91].

A robotic system is made of a set of connected physical devices (robots, sensors, peripheral devices). The related controller, i.e *all the computer resources which are required for the on-line control of the system*, should be open and modular. This controller handles

- a communication system which basically takes into account the distributed and real-time characteristics of the system and the controller itself ;
- a smart man-machine interface which integrates both design and validation tools, while allowing an access dedicated to the various system levels according to the type of concerned user.

We shall use the following terminology.

- an *application* is the set of all the operations performed by a robotic system in order to achieve a given objective ;
- a *module* is a set of boards and interfaces which communicate through a bus. Modules are connected by a communication system, generally a network of field-bus type ;
- a *module-task* is a real-time task (process) resident in a module. It is for example a basic element for realizing a complete control law. The coding of a finite state automaton representing an application is another example of module-task ;
- a *robot-task* is associated to the implementation of the global control law related to the set of physical resources needed for the achievement of an elementary operation. It corresponds to the interconnection of several module-tasks. It is a key issue of the proposed approach, and we will describe it in depth later.

The general architecture of the system which supports these elements (cf figure 1, taken from [Simon e.a.91]) has to be accessible at three levels using dedicated tools:

1- *system level*

- access: designer and fitter of the robotics system ;
- functions: interface realization, execution machine configuration, code distribution and execution control on the computer resources, access to operating system and communication primitives ;

2- *control level*

- access: control designer in charge of the realization ;

- functions: description of module-tasks, definition of their interconnection, specification of the temporal constraints issued from control schemes (allowed delays, sampling rates), library creation, coherence verification and validation ;

3- *application level*

- access: end user of the robotic system ;
- functions: specification of the application and of its temporal features, library creation, generation of an application program and supervision.

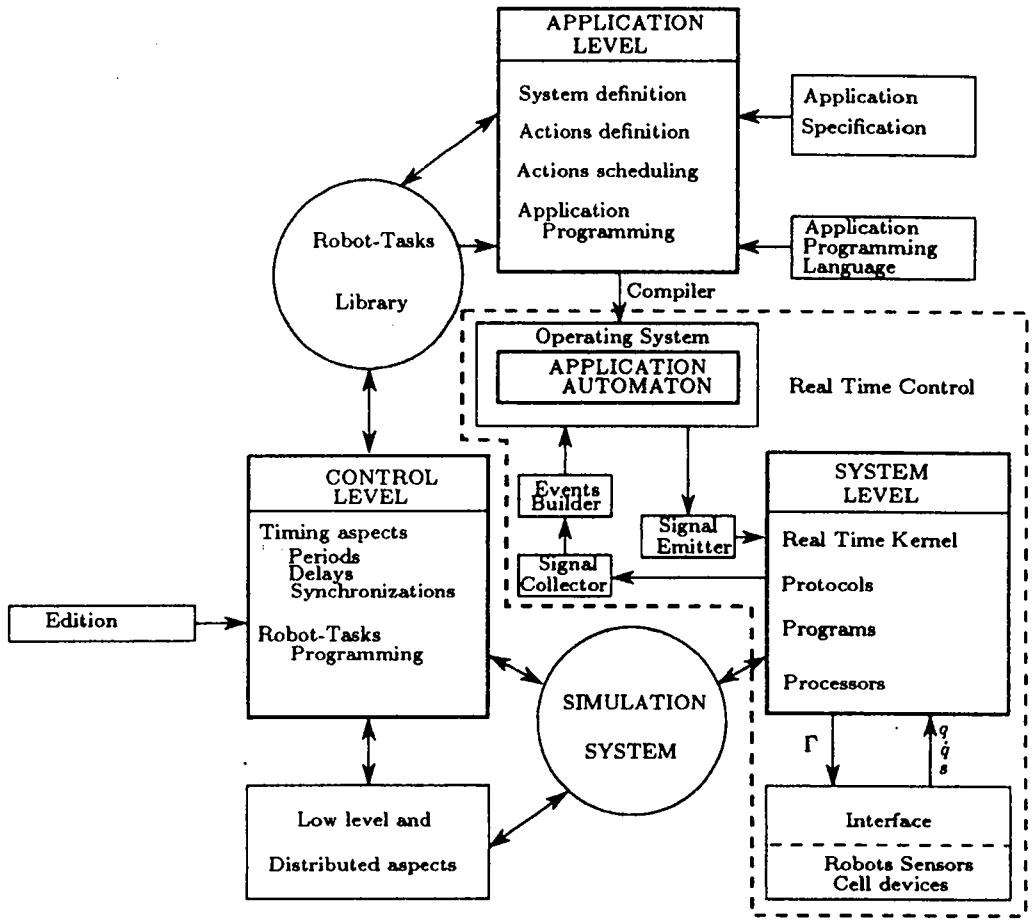


Figure 1: General Architecture of a Robot Control System

These three levels are presently at various states of design, implementation and validation. We only study the application level in the present report, knowing that the required information on the others is available, for example in [Simon e.a.91].

1.2 Some Generalities on the Proposed Approach

Thus, an application is roughly defined by a set of scheduled robot-tasks. It may be considered as a kind of Discrete Event System, since its sequencing depends on events, which may be external or internal to the control system (see for example [Brandin e.a.91]). The aim of this paper is to describe a way of modelling and programming applications defined in that way. As an application example, we consider all along this report a small flexible assembly cell such as the one presented in [Coste-Manière89]. It is of course a simple illustration of the proposed approach, and we clearly do not claim that this work is a contribution to the domain itself of the automated assembly.

In a quite simplified way, the process which starts from the specification of such an application and ends with the generation of some automaton characterizing its detailed behaviour might have the following form.

- *step 1* Generation of an assembly graph (as in [Gasmi90] or [de Mello e.a91]) ;
- *step 2* Expansion into a functional scheduling of the application (set of services requirements with temporal or logic constraints) ;
- *step 3* Scheduling and resources allocation with respect to results of step 2, and according to a given optimization criterion ;
- *step 4* Generation of an executable program, the lowest (sometimes called "atomic") level of which is an elementary action of the robotic system.

The main contribution of this work lies at the last step, even though some incursions at steps 2 and 3 are needed in the modelling stage. Indeed, we will not for example wonder how an optimal scheduling may be obtained at step 3.

It is now of some interest to precise what are the main options which motivate the choices done in the described approach. A classical concern for software quality was taken into account through five criteria:

1. Reliability of the resulting behaviour
2. Modularity
3. Easy evolutivity
4. Efficiency
5. Adequation of the man-machine interface

Point 1 was a major one. To reach this goal, the choice of *full determinism* was made. This means that the whole system behaviour is specified off-line, and this excludes for example on-line planning except if all the steps are reconsidered. It should be however emphasized that known exception handling processes may be a-priori included (cf section 2.2.6 and 3.6) ; for example, in order to compensate for a possible resource failure, an alternate solution may be proposed for every critical resource. This allows a kind of flexibility while limiting the risk of combinatorial explosion. An immediate consequence of this choice is to authorize the use of a purely synchronous language dedicated to programming of reactive systems, i.e here ESTEREL (cf appendix A), in

which determinism is a basic feature. The possibility of easily expressing parallelism, the generation of an automaton about which reasoning is possible are then advantages satisfying criteria 3 and 4.

Let us also emphasize that full determinism does not imply an open loop behaviour. In fact, it will be seen later that a particular attention is paid to the use of sensors, for task activation and monitoring as well as in the design of feedback controls.

Points 2 and 5 obviously suggest the use of an object-oriented approach. It will be therefore possible to describe a robotics system structurally while lying close to physical or fonctional entities, and designing in a modular way a set of data and related algorithms. As shown later, such an approach can also be used at a less classical level, i.e the design of a robot-task, by structurating under object form all the control components.

The general idea proposed in this report therefore consists in making these two aspects cooperating. At the modelling level, we privilegiate the study of the object 'robot-task', which will be given with a generic (local) behaviour encoded through a synchronous language. The application skeleton, that is to say the (global) plan representing logical and temporal dependancies of all the instances of these objects, including parallelism issues, will also use a synchronous language as an output. In that way, local and global behaviours can be fused in a coherent manner, and the production of a global application automaton may be easily achieved.

The report describes a preliminary study of some basic issues of the proposed approach. It is organized as follows. In the next section, the used object-oriented model is precised, including the key object, the robot-task. Then, in section 3, we present the elements of the programming language and a first implementation. Section 4 describes briefly some issues on the required interface between a synchronous language and an asynchronous world, followed by the presentation of some examples. Perspectives are drawn in the conclusion. Four Appendices are provided.

2 Towards Object-oriented Design of Robot-Tasks

The design of a robot task is a rather complex process. It involves several entities, physical or not, which may fortunately be structured in a natural way. Choosing an object-oriented approach allows coherence between these entities to be ensured, and improves the readability at every level in the design stage. It will also allow the realization of a well-suited man-machine interface

In a first step (the concept will be progressively refined in the following), a robot-task may be considered as the association of a sometimes complex control scheme, dedicated to given physical devices, and of a local behaviour related to events generated by a set of validation and monitoring signals. An object-oriented model should therefore take into account both the control scheme synthesis and the description of the involved physical entities. We will mainly detail the first issue, less classical.

2.1 Control Design and Task Functions

2.1.1 Overview

Let us consider a rigid robot with n joints, the position of which is denoted as a vector q. It is known that its state equation is given by its dynamics:

$$\Gamma = M(q)\ddot{q} + N(q,\dot{q},t), \dim(q) = \dim(M) = n \tag{1}$$

where:

- Γ is the vector of applied external forces (actuator torques),
- M is the kinetics energy matrix,
- N gathers gravity, centrifugal, Coriolis and friction forces,
- q and $\dot{q}$ constitute the natural state vector of the system.

It may be shown ([Samson87], [Samson e.a.91]) that an efficient way of specifying the objective that an user wishes to reach with the robot consists in defining an output n-dimensional C^2 function, $e(q,t)$, called *task function*, to be regulated to zero during a time interval $[0,T]$, starting from an initial position q_0.

Once it has been verified that the problem is well-posed, i.e that it exists a C^2 solution to equation $e(q,t) = 0$ and that some regularity conditions of the *task-jacobian* $J_T = \frac{\partial e}{\partial q}(q,t)$ are satisfied, it remains to find Γ. We may write:

$$\dot{e}(q,t) = \frac{\partial e}{\partial q}(q,t)\dot{q} + \frac{\partial e}{\partial t}(q,t) \tag{2}$$

and

$$\ddot{e}(q,t) = \frac{\partial e}{\partial q}(q,t)\ddot{q} + f(q,\dot{q},t) \tag{3}$$

with:

$$f(q,\dot{q},t) = \begin{bmatrix} \vdots \\ \dot{q}^T W_i(q,t)\dot{q} \\ \vdots \end{bmatrix} + 2\frac{\partial^2 e}{\partial q \partial t}(q,t)\dot{q} + \frac{\partial^2 e}{\partial^2 t}(q,t) \tag{4}$$

where $W_i(q,t)(i=1,\cdots,n)$ is the partial derivative of the i-th row of $\left(\frac{\partial e}{\partial q}\right)^T(q,t)$ with respect to q. It then appears that equations (1) and (3) may be combined, which leads to:

$$\Gamma = M\left(\frac{\partial e}{\partial q}\right)^{-1}\ddot{e} + N - M\left(\frac{\partial e}{\partial q}\right)^{-1} f \tag{5}$$

Then a control which ideally decouples and ensures a linear behaviour of the error e is:

$$\Gamma = M\left(\frac{\partial e}{\partial q}\right)^{-1} u + N - M\left(\frac{\partial e}{\partial q}\right)^{-1} f \tag{6}$$

The subcontrol u is generally a PD feedback of the form:

$$u' = -kG\left(\mu De + \dot{e}\right) \tag{7}$$

G and D being positive matrices, k and μ being positive scalars, all to be tuned by the user.

The ideal control scheme (6) (7) requires a perfect knowledge of all its components, which is neither possible, nor even wished. A more realistic approach consists in generalizing the previous control as:

$$\Gamma = -k\hat{M}\left(\widehat{\frac{\partial e}{\partial q}}\right)^{-1} G\left(\mu De + \widehat{\frac{\partial e}{\partial q}}\dot{q} + \widehat{\frac{\partial e}{\partial t}}\right) + \hat{N} - \hat{M}\left(\widehat{\frac{\partial e}{\partial q}}\right)^{-1}\hat{f} \tag{8}$$

where the carets point out that models (approximations, estimates) are used instead of the true terms. In this general expression, all the terms but μ, D and G are allowed to be functions of q and t, even of $\dot{q}$ for $k, \hat{f}$ and $\hat{N}$. The control (8) includes most of existing schemes: computed torque, resolved motion rate or acceleration control, indirect adaptive control,...

Among the conditions which suffice to ensure that the system (1) under the control (6) (7) is stable (robustness), we may select ([Samson e.a.91]) *gain* conditions (for example $k(.) > k_{min}(.) > 0$) and *modelling* conditions (the most important is for example $(\frac{\partial e}{\partial q})(\widehat{\frac{\partial e}{\partial q}})^{-1} > 0$ in the sense of matrix positivity).

2.1.2 Classes of Goals and Controls

Equation (8) gathers a large number of control possibilities and choices of user's goals. In a simplified approach, these two aspects may be examined independently, although it is clear, for example, that a demand of high performance or accuracy at the task level induces special requirements at the control level (well-fitted models or high gains). Classifying such sets of controls and tasks is therefore done using an object-oriented approach detailed in the next subsection. We will only give in the following some brief non-structured informations about the various features wich respectively charaterize tasks and controls.

Tasks as User's Goals

A task is mainly defined by the triplet (e, q_0, T). The dimension of e, n, is determined by the physical system (one or several robots, sensors) which will 'implement' it. Among the several existing possibilities, let us briefly present four main examples of task functions:

- Trajectory tracking in joint space: $e = q - q_r(t)$, where $q_r(t)$ is a desired trajectory in the joint space. Control will also need $\dot{q}_r(t)$ and $\ddot{q}_r(t)$ or some approximates of them.

- Trajectory tracking in another space: case of $\mathbf{SE_3}$ (the configuration space of frames or rigid bodies). Once a working frame with origin x is specified, and a parametrization E_a of the attitude error is selected, the task function may be written as:

$$e = \begin{pmatrix} x(q) - x_r(t) \\ E_a(q,t) \end{pmatrix} \tag{9}$$

  As previously, control may need first and second derivatives of e.

- Redundant task ([Samson e.a.91]). Its general form is:

$$e = W^{\dagger} e_1 + \alpha (I - W^{\dagger} W) \frac{\partial h_s}{\partial x} \tag{10}$$

  where:

  * e_1 is a first task vector, called main or primary task vector: it is for example an array of sensor signals to be controlled (proximeters, range finders, force or visual sensors ([Rives e.a.89])).

  * W is a matrix function such that Range $(W^T) \approx$ Range $(\frac{\partial e_1}{\partial x})^T$, '$x$' representing any reasonnable n-dimensional working space, for example $\mathbf{SE_3}$ with a frame linked to the robot end effector. $W^{\dagger}$ is the pseudo-inverse of W.

  * α is a positive scalar

  * h_s is a 'secondary' cost function to be minimized. h_s is aimed to represent for example one of the above given trajectory tracking objectives. In that case, and if e_1 is made of sensor outputs, the task is called 'hybrid task'.

- Optimization task. Let $h(q,t)$ a cost function to be minimized with respect to q. A related task function is then the gradient $e = \frac{\partial h}{\partial q}$

Many other cases exist ; it would also be necessary for the above cited tasks to go down to further detail levels, as done later. The reader would however easily imagine from these examples that this approach gathers very large possibilities and that the specification of an application requires several choices (type of function, parameter selection...) to be made.

Controls

A control is firstly the association of a task and of some modelling selections. At this step, we find for example the following choices ([Samson87]):

- Models of the kinetics energy matrix: for example constant, diagonal, computed through a dynamic expression $M(q)$, estimated on-line with various parametrizations, etc.

- Models of $\frac{\partial e}{\partial t}$ and $(\frac{\partial e}{\partial q})^{-1}$ (for example $(\widehat{\frac{\partial e}{\partial q}})^{-1} = (\frac{\partial e}{\partial q})^T$, $\widehat{\frac{\partial e}{\partial q}} = J_6$, the robot Jacobian, $\widehat{\frac{\partial e}{\partial q}} = I_n$, ...).

- Model of the expression $N - M(\frac{\partial e}{\partial q})^{-1} f$, linked or not to the previous choices.

In order to form the control equation (8), some parameter choices are also required: k (constant or non-linear), $\mu, G, D \ldots$

As previously, the possibilities are therefore numerous. Moreover, the control expression is in practice evaluated digitally ; it is therefore necessary to specify the computation periods for every term (these periods may differ, since for example updating of $\hat{M}$ and $\hat{N}$ does not need to be performed at the same rate as the one of u). Finally, checking that no disfunction occurs in the system requires the on-line monitoring of some relevant variables (reaching of joint limits, $||e(t)||$, $||\frac{\partial e}{\partial t}||$, $||\frac{\partial e}{\partial q}||$, det $(\frac{\partial e}{\partial q})$). A watchdog will also verify that the time limit T has not been reached prematuraly, and an evaluation of the positivity of $(\frac{\partial e}{\partial q})(\widehat{\frac{\partial e}{\partial q}})^{-1}$ will be a stability indicator.

All these elements will allow to define the class hierarchy of the controls we are concerned with.

2.2 The Object-oriented Model

In the proposed modelling of the robotics system, we may distinguish two main types of entities:

1. the entities of *algorithmical* kind (in a wide sense): those are all the components needed for synthetizing a control of type (8) and in the design of a robot-task.
2. the entities of *physical* kind: those are for example the devices which constitute a flexible assembly cell, or the workparts themselves ;

Of course these last entities operate *in fine* on physical devices, so there exists dependence relations between the two types. For each of them, we have selected some essential classes:

1. algorithmical entities:
   - *task functions*
   - *trajectory generators*
   - *models*
   - *controls*
   - *observers*
   - *robot-tasks*
2. physical entities:
   - *physical resources*
   - *parts*
   - *services to be provided by the resources.*

Dependences (in the sense of communication needs, or even, if required, multiple inheritance) between these classes are precised figure 2.

Let us now briefly present each of them, beginning with the key ones for our approach, that is to say the algorithmic entities. Details are provided in Appendix C.

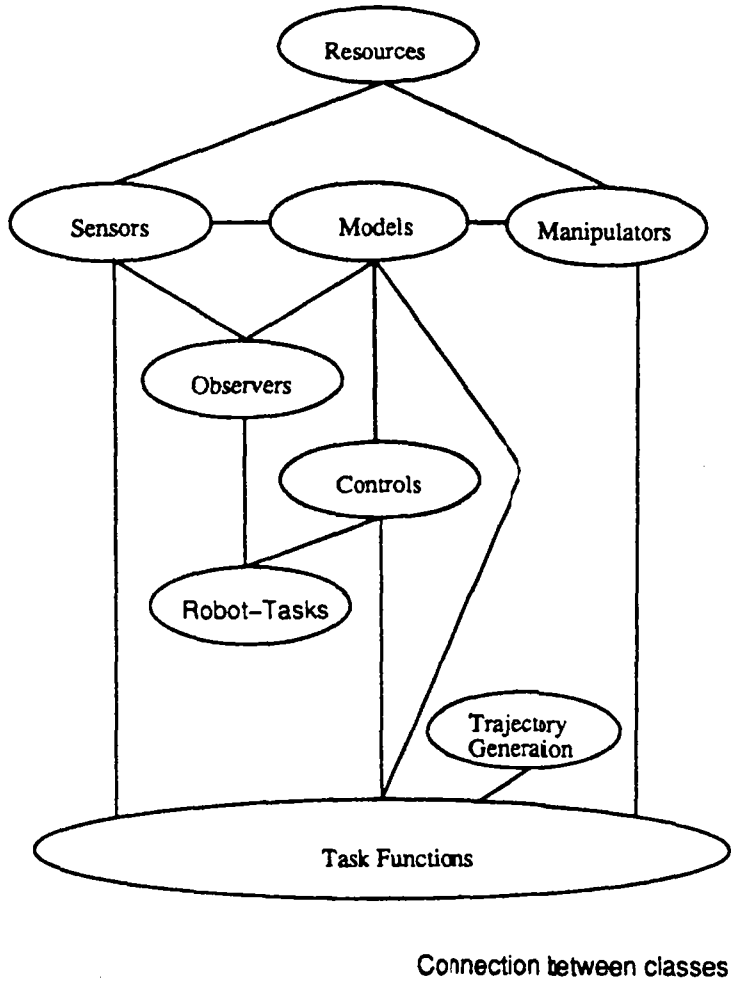


Figure 2: Connections between classes

Let us emphasize that in the proposed objects of the algorithmical group, we find two main kinds of methods:

- the methods which implement algorithms or computations: they are generally procedures in C, or even in assembly languages ;
- the methods which concern some local scheduling aspects of an application (in a wide sense, including parallelism and reactivity): they are encoded using ESTEREL, like in the *Robot-tasks* class.

2.2.1 Class *Task Functions*

Its class hierarchy is given figure 3.

An object instance in this class takes measurements and parameter values as inputs, and applies a computation method to them in order to produce the value of $e(q,t)$. A detailed example is treated in section 4.5. This class communicates, among others, with the subclasses *sensors* and *manipulators* of the *resources* class.

2.2.2 Class *Trajectory Generators*

Its class hierarchy is given figure 4.

It is aimed here to express one of the main user's needs, i.e. the tracking of a trajectory in some space by a robot or another device. Subclasses *polynomials* concern for example the use of spline interpolating functions between given points ([Dombre e.a.88]), while subclasses *reference models* allow some kind of dynamic behaviour to be specified. This class obviously communicates with the *Task Functions* class.

2.2.3 Class *Models*

Its class hierarchy is given figure 5.

It includes models linked to the physical system (task-independent, as the dynamical model) as well as models related to the task itself (various derivatives). It should be noticed that, for a *same* physical device, different models may coexist, according to the wished use. The main communication issues for this class concern *manipulators, controls* and *task functions* classes.

2.2.4 Class *Controls*

Its class hierarchy is given figure 4.

It mainly corresponds to the done modelling selections, which induce by analogy with the underlying minimization methods (cf [Samson e.a.91]) a behaviour of *Newton* (good trajectory tracking owing to second order terms) or *gradient* (first order terms only, but good robustness) types. The 'output' of this class is towards the *Robot-task* class.

2.2.5 Class *Observers*

Its class hierarchy is given figure 6.

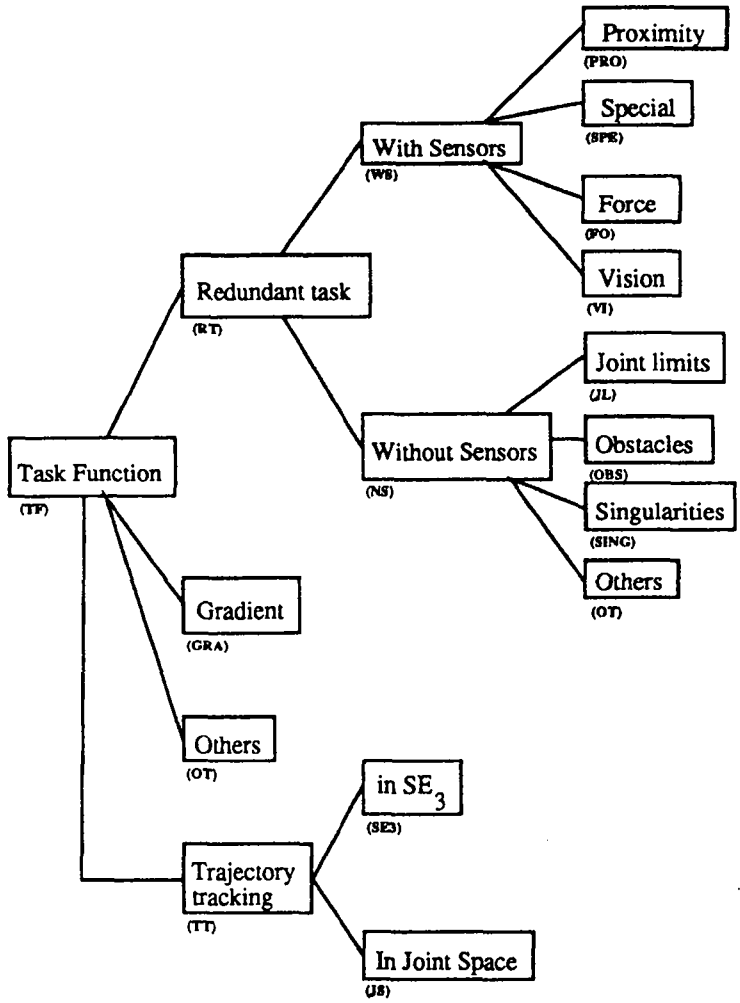


Figure 3: Class *Task Functions*

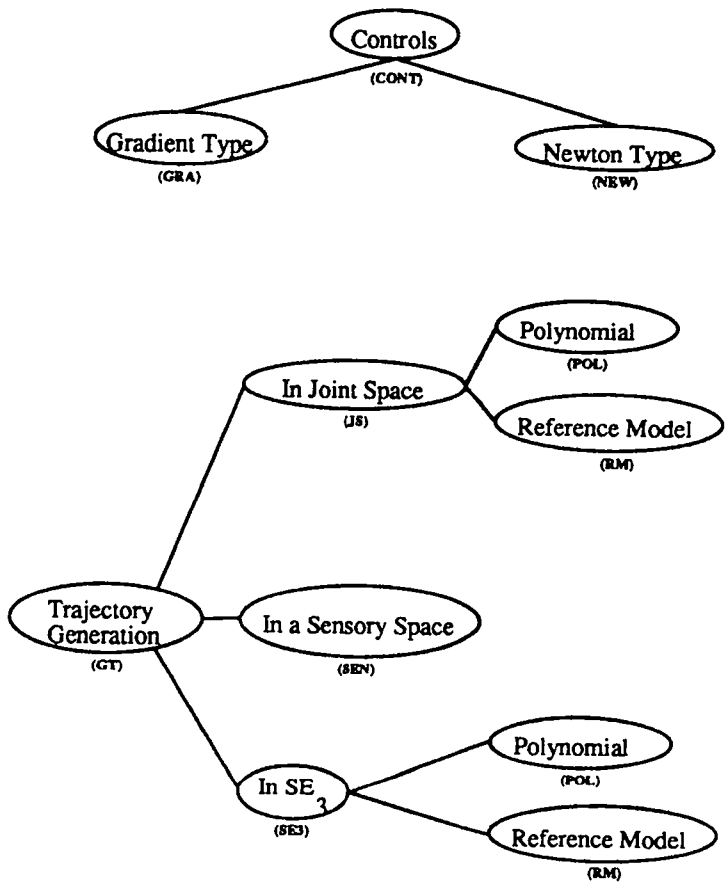


Figure 4: Classes *Controls* and *Trajectory Generators*

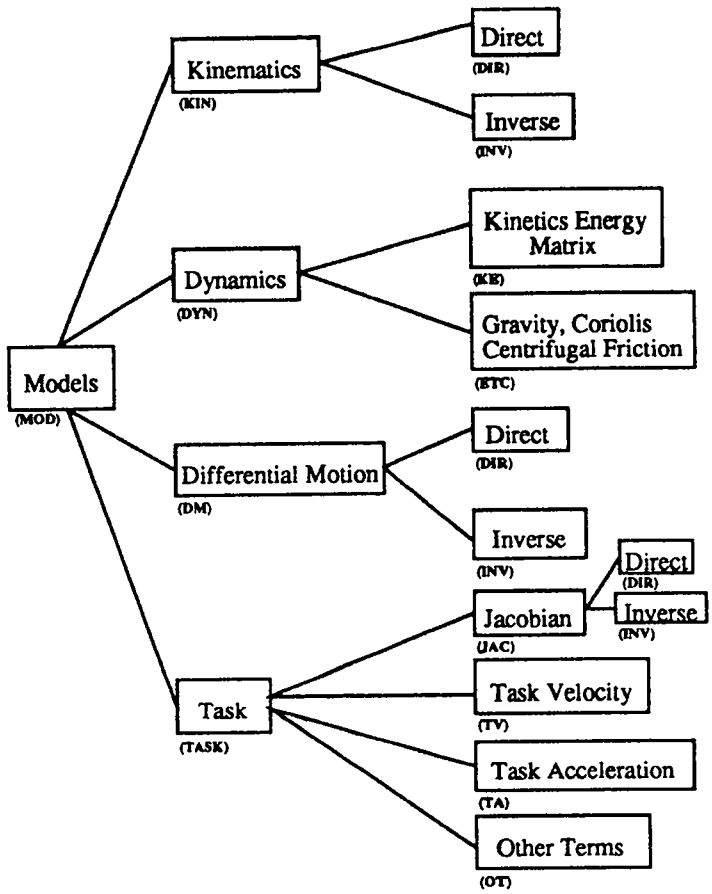


Figure 5: Class *Models*

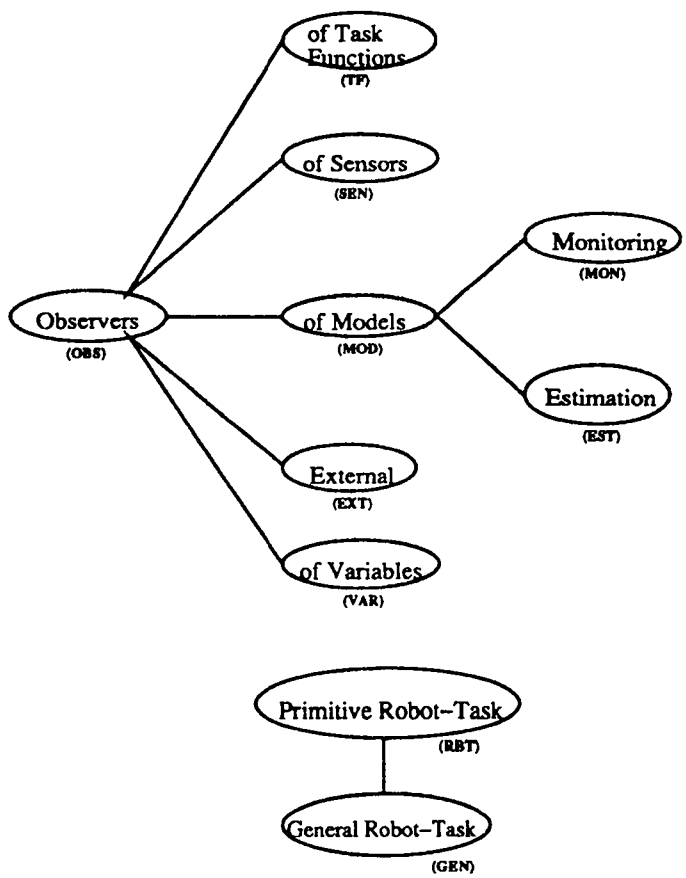


Figure 6: Classes *Observers* and *Robot-Tasks*

This class is concerned with quite various functions: for example, watching the elapsed time or monitoring the conditioning of a jacobian matrix. They are essential components of a robot-task, with the class of which observers communicates.

2.2.6 Class *Robot-Tasks*

Its very simple class hierarchy is given figure 6. Since this class is the essential one, we provide the reader with a detailed description of its generic object.

A primitive robot-task is characterized by the absence of internal event handling functions other than those which imply its ending. We give here a description of its very simple behaviour and a list of the various attributes and methods associated.

A robotics action is prompted by the outside world and its role is to respond to all the external inputs by performing transformations on them in order to produce outputs. Therefore, when defining a robot task, three fondamental aspects must be taken into account:

- the functional aspect (user's goal) ;
- the environmental aspect (communication with the environment) ;
- the behavioural aspect (types of reaction).

The first one is specified through a control expression. Let us give some details on the other aspects. Studying the environment consists in giving a list of all the events involved with the task execution. These events can be classified into three groups: the *pre-conditions*, the *post-conditions* and the *observers*. They can be either emitted or received during the execution of the task, according to their functionality. They are linked to specific actions execution. These actions correspond to the functional aspect of the task and include control actions, observations actions and error recovery actions. Once the actions and the signals are well known, it is necessary to organize their execution, that is to say to define which action has to be taken upon reception of a particular signal, and when is this action due to begin. The internal behaviour of the task is expressed using an ESTEREL program. Indeed a Robot task may be seen as a complex system made of a transformational part and of a reactive one according to the definition given by Harel and Pnueli in [Harel e.a.85].

Finally, the main Robot-tasks features are:

- the needed *pre-conditions*: a task can not be started until all the pre-conditions of the system are satisfied. The management of these signals is of prime necessity. When giving an implementation of the task, it is therefore necessary to verify throughout the use of sensors that the required pre-conditions are satisfied. This verification can be made through the activation of an observation task, or through the awaiting of signals. For example, before starting a trajectory-following task, the required pre-condition is: THE_TRAJECTORY_IS_FREE. The management of the pre-condition signals is detailed in the last item, throughout the description of the observers: pre-conditions (and post-conditions) are in fact particular observers.

- the *post-conditions*. Their role is double: they allow the system to check out that the task is completed, and they may be also considered as pre-conditions for others tasks. In any case, they can be emitted by the task itself, or the information can be obtained through sensor reading. In the previous example of trajectory following, the task is stopped when the target point is reached: thus, TARGET_POINT is the important signal.

- the *control expression* is linked to the functional aspect that is to say to the control objective and to the output function to realize. Its building up is complex and requires the use of the various models stated in this section, (the task functions and the controls models). The control procedure can be seen as the main part of the robot-task. The control procedure is a method of the object "robot-task" and must be executed using a new ESTEREL primitive, the **exec** statement. In the above mentioned example, the control consists in performing a trajectory following in a given space.

  Let us notice that in some cases, the control expression may be quite simple: this occurs for example when activating a conveyor belt, incerting a tool or asking for a specific sensor measurement. The related robot-task is then trivial.

- the *observers:* the execution of the robot task must be monitored. Observation functions are thus activated in parallel. Their role consist in watching a set of fundamental signals linked to the control execution. As said before, pre-conditions and post-conditions may be considered as particular observers. Other signals associated to the control algorithm itself have to be handled: for instance, an observer can be associated to the regularity condition of the task Jacobian. Some observers can be defined in order to detect failures during the execution. These signals can be forecasted in the design stage in order to allow an off-line specification of the robot task behaviour.

  Processings must be defined, to undertake actions allowing error recoveries. The possible processings associated to observers are:

  - type 0 processing: It is dedicated to pre- and post-conditions signals. Two types of conditions are considered: those always accessible because the associated sensor is always active, and those which require the activation of a particular sensor. Once the signal is received, the reading or the evaluation of the sensor value is stopped and the next actions are executed sequentially. **Waiting** (or reading) and **Evaluation** are special primitives described in section 3.4.2.
  - type 1 processing: the failure indicated by the received signal requires an adequate processing performed without interrupting the task. For example, reception of the JACOBIAN signal means the proximity of a singularity, which may be overcome using a procedure given in [Samson e.a.91]. A slight change of the parameters must be performed, and stopping the task is unnecessary. This is an internal modification of the task.
  - type 2 processing: the task may no longer be achieved, because no internal recoveries are possible. Upon reception of a critical signal, the task must be killed and instead another one must be executed. For instance, when a

manipulator has reached a JOINT_LIMIT the task function must be changed, for example by selecting another trajectory generation.

- type 3 processing: the detected error is critical for the execution of the whole application. In this case, a controlled system shutdown must be performed and the operator's skill is needed.

The incidence of these various kinds of processings at the application programmind level is evoked in section 3.6.

Remark: type 3 processing is particularly important when considering the last implementation stage of a complete robotics application.

At this step of the description, a generic robot-task example may be provided. The skeleton of a contour following robot task is given in section 4.5: let us here consider a task TASK_EXAMPLE, the objective of which is to perform a control: CONTROL_EXAMPLE, if and only if two pre-conditions are met: PRE1 and PRE2. In order to supervise the execution of CONTROL_EXAMPLE two observers are activated: OBS1 and OBS2 and the possible processings associated are PROCESSING1 (of type 1) and PROCESSING2 (of type 2, i.e. the task is killed). The description will be completed with two post-conditions: POST1 and POST2.

Before giving the skeleton of the robot-task, in an ESTEREL-like syntax, we need to present the **exec** statement. Executing a control algorithm is a non instantaneous action which requires a certain amount of time, generally unknown. Considering the trajectory following example, an incorrect solution could be to use the ESTEREL primitive `call` dedicated to procedure execution. But this execution should be instantaneous, and therefore is not to be used here. Another solution could be to define an external task, TRAJECTORY_FOLLOWING prompted by two signals. It is activated with a start signal. An end signal is emitted by the task, i.e. is awaited by the ESTEREL automaton when the task is completed.

This could be translated in ESTEREL as follows:

```
emit   START_TASK;
await  END_TASK;
```

This expanses a `call` statement and solves the task execution duration problem. However, a last problem remains. Let us consider an exception handling: e.g. the task may no longer be executed when an obstacle is encountered. It must therefore be killed instantaneously, and a kill signal must be handled.

The need for managing all these signals is a strong constraint which is incompatible with modularity. Moreover, it can not be controlled in a rigorous way unless it is carried out by the compiler itself. The **exec** statement was therefore designed for asynchronous tasks management. It is based on a rigourous semantics.

We may now consider the execution of the task: it starts with the awaiting of all the pre-conditions. This is performed in parallel. We do not detail the waiting of the pre-conditions and refer the reader to section 3.4.2 to find out the possible expansion of this primitive. Once all the conditions are satisfied, the control algorithm and the associated observers will start in sequence.

Using the **exec** statement, the control algorithm and the observers are activated in parallel. The processing execution (also activated using **exec**) depends on the result of the observation functions. Control, processings and post-conditions are part of a declared exception (**trap** statement), and the exception is raised (using the key word **exit**) when post-conditions are received: the task execution is over.

We give a very simple task skeleton in a mixed syntax close to the natural language and to the ESTEREL one:

```
[
 <waiting_for (PRE1)>
 ||
 <waiting_for (PRE2)>
]
;
trap ROBOT_TASK_SKELETON in
 exec CONTROL_EXAMPLE()()
 ||
 loop
  <waiting(OBS1)> ;
  exec PROCESSING1 ()()                    % type 1 processing
 end % loop
 ||
  <waiting(OBS2)> ;
  exit ROBOT_TASK_SKELETON % this corresponds to
                           % a type 2 processing: the task must be killed
 ||
 [
  [
   <waiting(POST1)>
   ||
   <waiting(POST2)>
  ]
  ;
  exit ROBOT_TASK_SKELETON
 ]
end % trap
```

2.2.7 Classes Concerned with Physical Entities

The main link of these classes with the previous ones, in particular the robot-tasks, may be done through the concept of *service*: a service is an activity being exerted by a resource of a set of cooperating resources. A service may be decomposed in a set of synchronized elementary services (called also atomic actions by other authors) strongly related to robot-tasks. Every elementary service modifies the *state* of a resource and/or a part.

In the context of automated assembly, the main concerned classes are:

- *the physical resources*
- *the services*

- *the parts* (a part is a physical object associated at any time with a resource ; its state is known, maintained and accessible t any request)

to which may possibly be added other objects linked to the application domain (cf [Gasmi90]): cells, assembly graphs... However, we did not examinate all these aspects in depth, because they are not our main concern, and that several related works are reported in the literature ; so there is very little originality in this part of the approach, except the association which might be made between services and robot-tasks.

Figure 7 gives the class hierarchy of the *resource* class. In the example of assembly, the main services achievable by these resources are:

- *Feeding*
- *Processing (contouring, surfacing, welding, drilling, ...)*
- *Information/Measurement*
- *Part Transport/Manipulation (i.e. change of frame position)*
- *Motion (small/large ; fine/unaccurate)*
- *Storage (intermediary buffer/initial or final)*
- *Grasping/ungrasping (in open loop/sensor-based)*
- *Reception (passive/clamped)*
- *Assembly (passive/sensor based ;cooperative)*

Their assignment to resources is given in the following table entitled "Services and Resources".

	Feeding	Processing	Information	Transport	Motion	Storage	Grasping	Reception	Assembly
Feeders	x								
Removable devices		x	x	x			x		
Buffers						x			
Conveyors	x			x	x				
Manipulators				x	x				x
Assembly Supports			x		x			x	x
Machine-tools		x							
Sensors			x						

The way of expanding a service in a set of elementary actions associated to robot tasks is not treated here. It might however be expressed in a coherent way with the application langage described in section 3. For example, the grasping of part P on the buffer B by the robot R might be expanded as:

```
REACH (B)
WHEN (Free-access (B) and Available (P))
    SENSOR_BASED_REACH (P) ;
    FORCE_CONTROLLED_GRASP (P) ;
    QUIT(B)
```

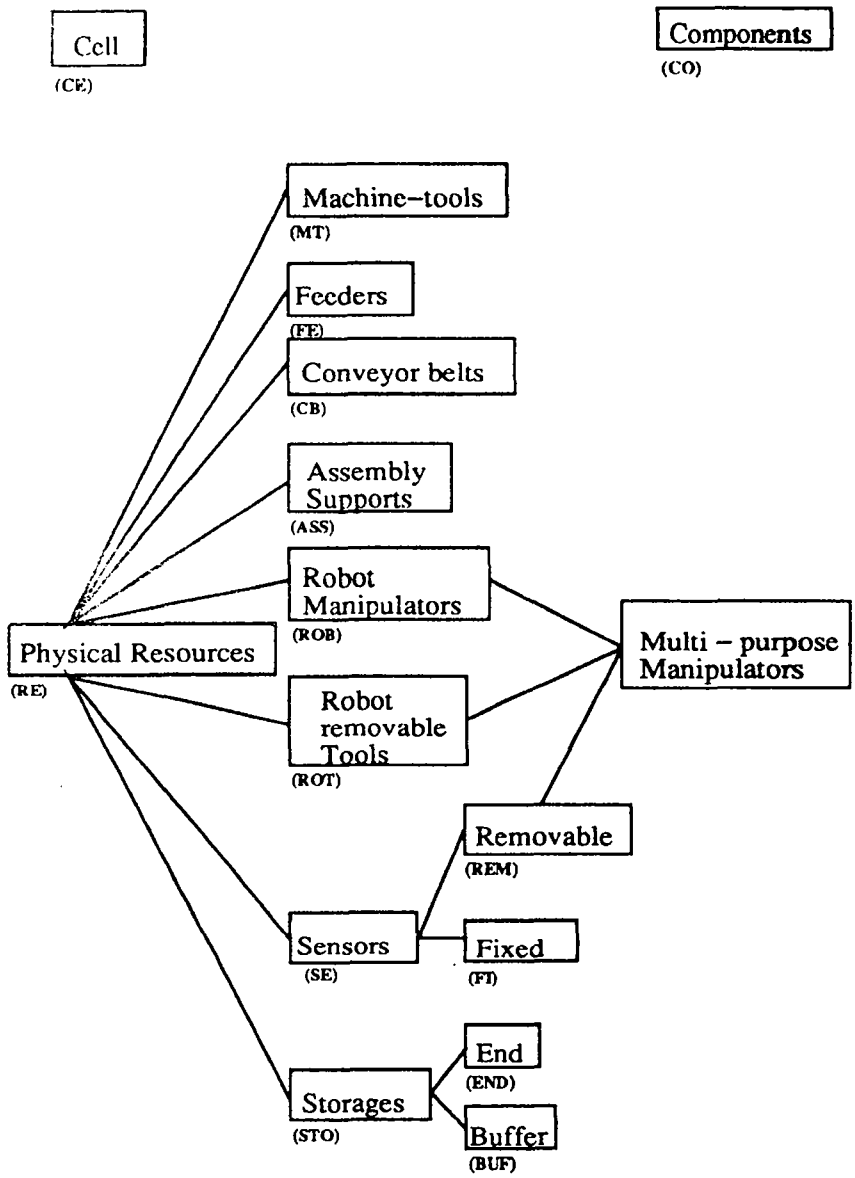


Figure 7: Class *Physical Resources*

2.3 Some Comments

The approach we followed to design the proposed methodology was, as in [Borrelly e.a.90], of *'bottom-up'* type. The analysis indeed started from the lowest level (module-tasks structure), which offers a reasonable chance of feasibility. So, the approach is implementable even if higher steps are missing. For example, a robot-task might be in the future automatically defined instead of being designed by an operator, without implicating other levels to be modified. A *'top-down'* analysis would certainly have implied difficulties of taking efficiently into account some critical features related to real-time or automatic control constraints. Incidentally, this is the reason of several unsuccessful outcomes in the design of programming and control systems for robotics applications.

Let us now come back to robot-tasks. Once everything needed in a given robot-task has been instantiated, we get:

- a complete definition (in the sense of a continuous-time formal specification) of the control vector, from which may be deduced a block-diagram decomposition such as the one of figure 8 taken from [Simon e.a.91] and representing an example treated in section 4.5 ;

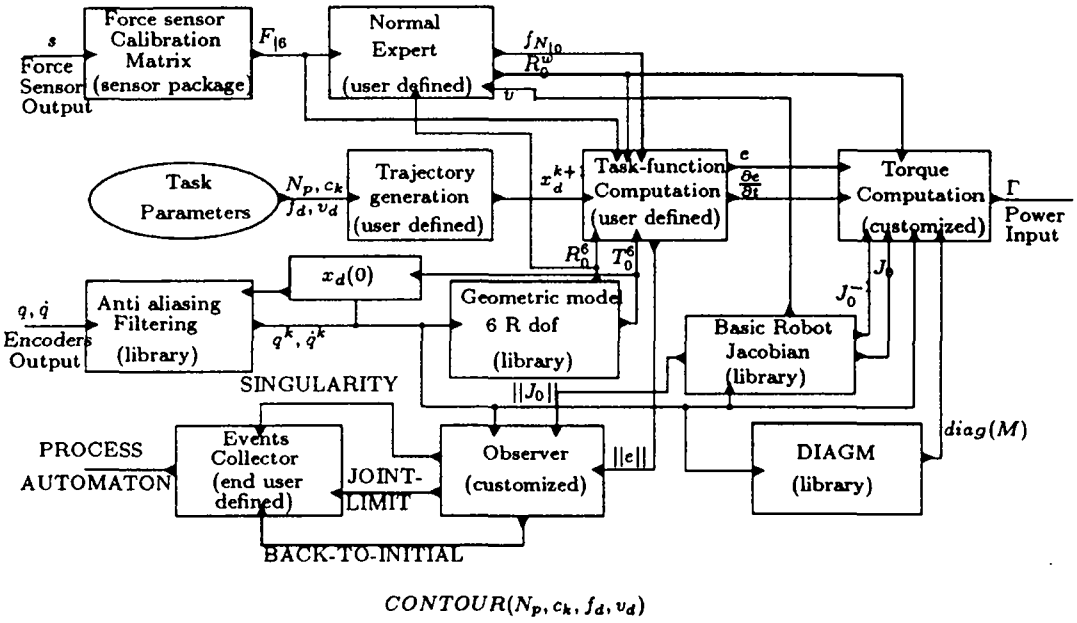


Figure 8: Task-module expansion of a robot-task

- a set of observers to be activated, with associated parameters ;
- a description of the sequencing and of the exception handling procedures *inside* the robot-task.

On the other hand, most of the module-tasks may be deduced from an object instantiation done at the design step. The block-diagram representation may therefore be built under a preliminary form as soon as the robot-task is fully defined. As emphasized in [Simon e.a.91], temporal specifications (sampling frequencies, maximum

allowed delays, synchronization types, etc...) should be associated to this diagram. Typed communication ports between modules have also to be precised. This further design step might certainly be performed interactively from the preliminary generated diagram, since it seems difficult to include such data at the previous step of formal specification.

3 An Application Programming Language

3.1 Introduction

In the general context of the Open Robot Controller previously discussed, an application programming language, allowing the task-level robot programming, i.e. the sequencing, or more generally the temporal arrangement, of the robot tasks is now needed. The use of ESTEREL, as described earlier, allows to specify the internal behaviour of robot tasks and their execution on the Open Robot Controller architecture, through a synchronous/asynchronous approach [Espiau e.a90]. For the specification of plans of actions, i.e. at the level of sets of tasks, a language provided with imperative control structures was designed [Rutten90], [Rutten e.a.90] and defined in terms of temporal logic, in relation with artificial intelligence planning.

The integration of all these aspects is done in the present approach by proposing a language inspired from the planning language, taking over most of its control structures and giving a translation of it into ESTEREL corresponding to their temporal definition, so that it is executable on the robot controller architecture.

This way, the user of the Open Robot Controller is offered a language enabling him to express a sequencing of primitive tasks without having to know ESTEREL, and more importantly without having to go into the details of the signals exchanges and dialogues for each particular arrangement of tasks. On the other hand, the abstraction level of task representation (owing to the use of preconditions and postconditions) is the same as that in A.I. planning, based on predicate logic and enabling some reasoning on the plans. This opens perspectives towards the connecting of an A.I. planner to the Open Robot Controller.

The primitives of the language are imperative: the control structures are used to build applications (or plans) of primitive tasks (or actions) in a clearly structured way: each control structure specifies a particular temporal arrangement of the actions or subplans in its scope. The execution then follows the imbrication of control structures. They are divided into several kinds:

- those that are temporally independent of the environment: sequence, conditional statement, parallelism ...
- the reactive ones: they describe the waiting for some condition and the execution of a reaction ;
- those that specify precedence or synchronization points between parallel plans. .

3.2 The language grammar

3.2.1 Syntax notation

We use a usual form for grammar definition, with the following conventions:

- $X ::= Y\ Z$ is read: X has the form of Y followed by Z. For example: *S::= declarations body* . `END`. means that an application S is made of *declarations*, followed by a *body* finished by a dot: ".", followed by an end-marker "`END.`".
- $X \mid Y$ is a notation for an alternative, and means: X or Y. For example: *body::= cascade | plan* means that a *body* is either a *cascade* or a *plan.*

- words in uppercase typewriter characters are reserved words of the language (for example: COND),
- braces enclose a repeated item, that may appear zero or more times. For example: *declarations* ::= { *decl* }). means that *declarations* is made of zero, one or more occurrences of *decl*.

3.2.2 Language syntax summary

The axiom for the grammar is S, from which an application can be derived:

S	::=	*declarations body* . END.
declarations	::=	{ *decl* }
decl	::=	PLAN *compound-task* : *plan* .
		| PRECED *prec-point* .
body	::=	*cascade* | *plan*
cascade	::=	CASCADE(*compound-task* { , *compound-task* })
plan	::=	*task*
		| *compound-task*
		| NOTHING
		| SEQ(*plan* { , *plan* })
		| PAR(*plan* { , *plan* })
		| COND(*condition* , *plan* , *plan*)
		| AS-LONG-AS(*plan* , *task* { , *task* })
		| UNTIL(*condition* , *task* { , *task* })
		| *loop-or-rule*
		| ASSOC(*plan* , *loop-or-rule* { , *loop-or-rule* })
		| UNTIL-R(*condition* , *loop-or-rule* { , *loop-or-rule*})
		| BLOCK(*prec-point*)
		| UNBLOCK(*prec-point*)
loop-or-rule	::=	*rule*
		| LOOP(*plan*)
rule	::=	WHEN(*condition* , *plan*)
		| WHENEVER(*condition* , *plan*)
condition	::=	NOT *condition-name*
		| *condition-name*
		| TRUE

where:

- *condition-name* and *task* are defined in the robot-tasks model,
- *compound-task* and *prec-point* are declared in the declaration statements (PLAN or PRECED).

From a lexical point of view, the possibility is given of inserting text for comments: any text contained between two % characters is ignored.

In the following, we will describe all the language primitives: first the basic ones in section 3.3, related to the robot, and enabling the declaration of an environment for the plan, then the control structures with which plans are constructed, in section 3.4. The use of the language primitives is illustrated in section 4.2.3 and the translation scheme is given in detail in section 3.5.

3.3 Basic features of the languages

3.3.1 The primitives defined in the robot model

The very basic primitives of the language, the *condition* and the *task*, correspond to procedures programmed at a lower level, closer to the execution architecture.

Conditions. They are boolean values to be acquired from the external execution environment trough sensors. Using the `NOT` operator, the negation of their value can be used. The constant `TRUE` has the meaning of an always verified condition, that does not require any test.

We are only interested in "all or nothing" sensors, because of the use we are making of this information at our level: what is needed is an information allowing us to choose one branch or another or to start an action, instead of a quantitative information used in a control computation (this latter being done at the lower levels).

We give ourselves two types of these conditions:

- the "cheap" ones, i.e. those for which the corresponding sensor is always active in the robot architecture: we then only have to read their current value.
- the "expensive" ones, i.e. those where a particular sensor has to be activated, with an operating procedure, and the value of which has to be awaited.

These conditions will be used in two different manners:

- evaluation, where the actual current value is needed in order to make a choice between several possibles branches ;
- waiting for the "true" value, since only this value terminates the waiting.

Robot-Tasks. As seen in the previous section, they are defined by:

- a set (or list) of preconditions,
- a set of observers with, for each of them, an associated behaviour (procedure),
- a set of postconditions,
- a procedure implementing the control (task function, trajectory generation ...)

Each precondition, observation and postcondition is a condition like those seen before. The control and related procedures are defined elsewhere.

As seen in section 2.2.6, an execution of such a task begins by waiting for all the preconditions to be true ; then the control procedure is executed until all postconditions have been met or more generally, untill any logical combination of postconditions is satisfied ; in the meantime, at each occurrence of an event generated by an observer, the corresponding processing procedure is executed.

3.3.2 The declarations

An application specification may begin with a list of declarations, followed by a *body* terminated by a dot ".", followed by the marker "END." (that terminates input file reading):

S ::= *declarations body* . **END.**

The declarations can be of two forms:

declarations ::= { *decl* }
decl ::= **PLAN** *compound-task* : *plan* .
| **PRECED** *prec-point* .

Compound tasks. The compound actions enable to specify re-useable parts of the application, as a kind of macro-task. They are declared by a **PLAN** statement, where the *compound task* is defined by the corresponding *plan*: i.e. when a *compound task* is encountered in the application, it is replaced by the text of the *plan*.

Precedence points. They are used as synchronizing points between parallel branches in an applications, especially for precedence: i.e. one can specify that one point in one part of the application must be executed before another point in another part of the application. For each of them, it will be possible to use the synchronizing instructions **BLOCK** and **UNBLOCK**, as explained further. They are declared in a **PRECED** statement.

3.3.3 The body of the application

The *body* is either a standard *plan*, as described in the next section, or a *cascade*.

body ::= *cascade* | *plan*
cascade ::= **CASCADE(** *compound-task* { , *compound-task* })

The *cascade* form behaves like a parallel construct, but uses a particular option of the ESTEREL compiler, allowing to obtain a strong optimization of the code produced, under the constraint that the parallel branches do not communicate with each other ; at our level, this means that no precedence points should be used in the application when the **CASCADE** construct is used.

3.4 The control structures of the language

3.4.1 Environment-independent primitives

A *plan* may then be:

a primitive or general *task* defined in the robot-task design step ; here it will have the form of a name, like **grasp** or **move_to** ;

a *compound task* , referred to by its name, as declared in a **PLAN** statement: the occurrence of the *compound task* in the application is replaced by the corresponding declared *plan* ;

nothing :

NOTHING

which does nothing and takes no time: it can be used in a conditional statement, if a branch must be empty.

a sequence :

SEQ(*plan* { , *plan* })

The plans in the list are to be executed one immediately after the other, in the order of the list ;

a parallel statement :

PAR(*plan* { , *plan* })

The plans in the list all start at the beginning of the construct, which ends with the latest terminating plan ;

a conditional statement :

COND(*condition* , *plan* , *plan*)

The *condition* is evaluated through a sensor, as defined in the sensor class. If it is true, the first *plan* is chosen, else the second one is ;

a loop statement :

LOOP(*plan*)

The execution of the *plan* is endlessly iterated. The execution of this construct has no termination of its own: it has to be determined from outside, either through the user interface, or by associating its termination to that of a parallel plan or an external condition. It can be used in the **ASSOC** and **UNTIL-R** constructs presented further ;

the association of a set of *tasks* to a *plan* :

AS-LONG-AS(*plan* , *task* { , *task* })

The *tasks* are executed as long as the *plan* is executed in parallel: i.e. they are started together and the termination of the *plan* causes that of the *tasks*, and of the construct ;

3.4.2 Reaction primitives

A *plan* can also be reactive:

the association of a set of *tasks* to a *condition* :

UNTIL(*condition* , *task* { , *task* })

The *tasks* are executed until the *condition* is encountered in the environment ;

a rule : two kinds of rules are available:

simple rule :

WHEN(*condition* , *plan*)

The *condition* is awaited to become true (through the sensor defined in the robot model), and as soon as it is true, the *plan* is executed in reaction. The termination of the plan is the termination of the rule. If the *condition* does not become true, the rule does not terminate ;

A plan for simply *waiting* for a *condition* can be written:

WHEN(*condition* , NOTHING)

repetitive rule :

WHENEVER(*condition* , *plan*)

corresponds to an endless iteration on a simple rule.

These constructs do not always terminate: their termination depends on the *condition* in the external world. The following constructs will allow to constrain the execution of rules tighter.

the association of a set of *loops* or *rules* to a *plan* :

ASSOC(*plan* , *loop-or-rule* { , *loop-or-rule* })

As long as the *plan* is executed, when or whenever some conditions are met, the corresponding reactions are executed, or the body of the loop is iterated. If a reaction is busy being executed when the *plan* terminates, then the ASSOC construct terminates only when the reaction is terminated ; i.e. there is no interruption of a reaction that has been fired, as in [Rutten90].

The difference with the parallel construct PAR is that, when the plan terminates, *loops* and *rules* that are waiting for their *condition* to become true are terminated by the construct, without waiting for a termination of their own.

the association of a set of *loops* or *rules* to a *condition* :

UNTIL-R(*condition* , *loop-or-rule* { , *loop-or-rule* })

The set of *loops-or-rules* is active (i.e. when or whenever some conditions are met, the corresponding reactions are executed) until the *condition* is met in the environment (through a sensor). As in the ASSOC construct, if a reaction is then busy, it is not interrupted.

3.4.3 Precedence and synchronization

Two instructions are provided for the use of precedence points:

blocking on a *precedence point* :

BLOCK(*prec-point*)

is a blocking statement, waiting until some parallel branch of the plan executes the UNBLOCK construct on the same *precedence point.* However, if this has been done before already, then the BLOCK statement, being immediately released, is not blocking ;

unblocking on a *precedence point* :

UNBLOCK(*prec-point*)

is a non-blocking statement, releasing all the other branches blocked on the same *precedence point* by a BLOCK instruction, or the next one that will be.

This way, the part of the plan preceding the UNBLOCK instruction is executed before the part of the plan following the BLOCK statement. For example, the control scheme illustrated in fig 9, which can not be written exactly using only PAR and SEQ constructs, is achieved in the following application, using a precedence point prec between P_2 and P_5:

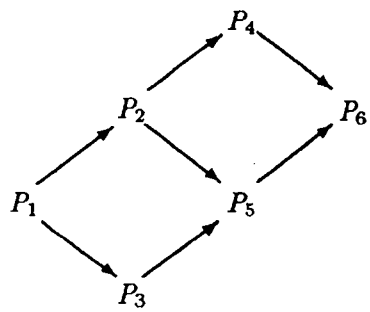


Figure 9: Control scheme requiring the use of a precedence point.

```
PRECED prec .

SEQ(   P1 ,
     PAR(  SEQ( P2 , UNBLOCK( prec ) , P4 ) ,
           SEQ( P3 , BLOCK( prec ) , P5 ) ),
       P6   ).

END.
```

In addition to the precedences entailed by the sequences, the execution of the sub-plan P_2 always precedes that of P_5.

Making a "Rendez-Vous" then consists in crossing two precedence relationships, using two precedence points as in:

```
PAR(  SEQ(P1, BLOCK( prec1 ), UNBLOCK( prec2 ), P2) ,
      SEQ(P3, UNBLOCK( prec1 ), BLOCK( prec2 ), P4) )
```

The use of the language is examplified in sections 4.2.3 and 4.4.2.

3.5 Translation into ESTEREL

The translation transforms plans written in the application language into ESTEREL code. The translator, an experimental version of which has been written in PROLOG II [Le Huitouze88], takes as input a file with a name suffixed by `.tf` (for the initials of TEMPUS FUGIT, as a recall of previous related work [Rutten90]). It produces a file with the same name, suffixed by `.strl`: this file is directly useable for a compilation using the standard `esterel` command, which produces in turn *C* code, itself to be compiled using cc.

3.5.1 Top-level ESTEREL module

The code produced as a result of the translation into ESTEREL is presented in the following, displayed in frames. It is a complete ESTEREL program, constituted by at least one module, the name of which is composed from the input *file name*:

```
module M_file name :
```

Then comes the declaration of input signals:

```
input
  STOP
  ,
  VAL_condition-name (boolean)
  ...
  ;
```

where STOP is a default signal allowing to accept a stopping signal from the outside of the program execution (e.g. from an operator console). Other input signals concern the conditions: for each condition *condition-name*, a boolean valued signal is declared, as above. An interesting feature of ESTEREL, the **sensor** kind of signal, corresponding to a sensor variable that can be read only (neither emitted nor received), can not be used here, because we want to be able to wait for the value to be true: therefore we must know when the value changes, which imposes the use of an **input** signal.

In the present version, no output signals are generated or used ; however, they could be used for communication with the operator, through a user-interface, for example.

Then come the ESTEREL **tasks**, that represent the external procedures, called through an **exec** statement (see section 2.2.6), corresponding to task command or sensor command functions:

```
task
  command () ()
  ...
  ;
```

These *commands* are often used in **trap** statements, which entails that, when an **exit** is executed, the execs are terminated, i.e. the corresponding *commands* are killed.

The behaviour of the robot is then determined at a lower level in the architecture. For example, such a visceral behaviour can consist in maintaining the current position, until a new *command* takes the control.

Then comes the local signals declaration: they concern dialogue signals for the management of *precedence points*, as will be explained further, with the `BLOCK` and `UNBLOCK` instructions.

The body of the ESTEREL module is composed of the translation of the plan. When *precedence points* are present, for each of them there is a part of the synchronization dialogues that is in parallel with the plan: this will be detailed further with the `BLOCK` and `UNBLOCK` instructions.

The translation of the body of the application, may then be done in two ways:

Standard plan. In the case of a standard plan, the ESTEREL program is made of one module, the body of which corresponds directly to the translation of the body of the application.

Cascade. This construct allows to use the ESTEREL compiler `-cascade` option. This option consists in compiling the ESTEREL program into several separate automata, the size of the resulting cascade of automata being smaller than the one of a single automaton, in particular if they have only little communication.

This is of some interest in our case, since:

it is relevant. As the example of section 4.2.3 illustrates, it is important to use external sensor information for the task sequencing. This information has more sense than internal synchronizations in the program using local signals: without the use of sensor information, an application is executed as in an open loop, whereas robustness requires an interaction with the environment.

it is efficient. We are precisely in the case where:

- The use of external information through input signals implies that there are few communications between the parallel branches (provided *precedence points* are avoided: they are the only instructions generating dialogues, which are forbidden by the compiler when using the `-cascade` option).
- The standard compilation produces an automaton recognizing all the interleavings and combinations of the input signals, that are precisely numerous in our case, causing an explosion of the number of states of the produced automaton.

The program has to be provided with a different structure, imposed by the ESTEREL compiler [EsterelV3]. The constraint is to have a main module, the body of which is constituted by only a parallel statement, containing only `copymodule` statements, and with restricted communication between these submodules (in particular: no dialogues). The solution for us is to impose that a `CASCADE` statement in an application contains only *compound tasks*, and that the use of *precedence points* between them is avoided.

Then, the translation consists in producing one main module for the `CASCADE` statement, and one module for each of the *compound tasks*. For example, an application in a file `appl.tf`:

```
PLAN CTASK1 :   body for compound task 1.
PLAN CTASK2 :   body for compound task 2.
CASCADE( CTASK1,
         CTASK2 ).
END.
```

is translated in:

```
module M_appl :
declarations
  copymodule M_CTASK1
  ||
  copymodule M_CTASK2
.

module M_CTASK1 :
declarations used in the body
translation of the body for compound task 1
.

module M_CTASK2 :
declarations used in the body
translation of the body for compound task 2
.
```

The translation of the plan is explained in the following, where:

$< P > \longrightarrow$ ESTEREL *code*

describes the *code*, displayed in a box, as produced for a plan P.

3.5.2 Translation of the basic features

Conditions. As seen before, conditions are used in two different ways:

evaluation, where the value is evaluated and used for branching. In the case of a *condition* acquired by a sensor driven by a procedure *sensor_function*, the code produced for its evaluation is:

<evaluation(condition)> $\longrightarrow$

```
do
  exec sensor_function () ()
watching VAL_condition;
if < neg > ?VAL_condition
```

In the case of a *condition* for which no sensor procedure has to be started:

<evaluation(condition)> $\longrightarrow$

```
if < neg > ?VAL_condition
```

If the condition is TRUE, then:

*<evaluation(*TRUE*)>* $\longrightarrow$ `if true`

If the condition is used in a negation (i.e. has the form NOT *condition-name*), then

$< neg > \longrightarrow$ `not`

else it is void.

The `then` and `else` branches are generated elsewhere, when translating a `COND` statement, as explained further.

waiting, where we wait until the value is evaluated as being true.

In the case of a *condition* acquired by a sensor driven by a procedure *sensor_function*, the produced code for its evaluation is:

<waiting(condition)> $\longrightarrow$

```
trap END_WAIT in
 exec sensor_function () ()
 ||
 every VAL_condition do
   if < neg > ?VAL_condition
     then exit END_WAIT
   end % if
 end % do
end % trap
```

In the case of a *condition* for which no sensor procedure has to be started:

<waiting(condition)> $\longrightarrow$

```
trap END_WAIT in
 loop
   if < neg > ?VAL_condition
     then exit END_WAIT
   end % if
 each VAL_condition
end % trap
```

If the condition is `TRUE`, then:

<waiting(`TRUE`*)>* $\longrightarrow$ `nothing`

a compound task CT defined by a plan P is translated in the code for the plan:

$< CT > \longrightarrow < P >$

nothing : the application language instruction `NOTHING` is equivalent to ESTEREL's:

`< NOTHING >` $\longrightarrow$ `nothing`

Robot tasks. The code is produced using the definition given in section 2.2.6:

$< T > \longrightarrow$

```
[
 <waiting(pre1)>
 ||
 <waiting(pre2)>
] ;
trap END_TASK in
 exec command ()()
 ||
 loop
   <waiting(obs1)>   ;
   exec t1 ()()
 end % loop
 ||
 loop
   <waiting(obs2)>   ;
   exec t2 ()()
 end % loop
 ||
 [
   [
     <waiting(post1)>
     ||
     <waiting(post2)>
   ]   ;
   exit END_TASK
 ]
end % trap
```

3.5.3 Plans

For each of the control primitives in the applications language, the following ESTEREL code is produced:

a sequence :

$<$ SEQ(*plan* , *rest of the sequence*) $>$ $\longrightarrow$

```
<plan>
;
< SEQ(rest of the sequence) >
```

We make use of the ESTEREL semi-colon (";") operator ; although the semantics of the language makes that it is not exactly a sequence operator, our use of it is correct, considering the sub-plans that are sequenced.

If the sequence is made of only one plan, the produced code is:

$<$ SEQ(*plan*) $>$ $\longrightarrow$ $<$ *plan* $>$.

a parallel construct :

$<$ PAR(*plan* , *rest of the parallelism*) $>$ $\longrightarrow$

```
<plan>
||
< PAR(rest of the parallelism) >
```

Here we use ESTEREL's parallelism operator ("||"), with its "co-begin, co-end" behaviour.

If the parallel construct is made of only one plan, the produced code is:

$$< \texttt{PAR(}\ plan\ \texttt{)} > \longrightarrow < plan >.$$

a conditional statement :

$$< \texttt{COND(}\ condition\ \texttt{,}\ plan_{true}\ \texttt{,}\ plan_{false}\ \texttt{)} > \longrightarrow$$

```
<evaluation(condition)>
then < plan_true >
else < plan_false >
end % if
```

As was said before, the `if` part of the ESTEREL conditional statement is produced elsewhere, namely in the translation of the *evaluation* of a *condition.*

a loop :

$$< \texttt{LOOP(}\ plan\ \texttt{)} > \longrightarrow$$

```
signal
  Q_LOOP,
  STOP_LOOP
in
  trap END_LOOP in
     loop
        emit Q_LOOP ;
        present STOP_LOOP
           then exit END_LOOP
        end % present
        ;
        < plan >
     end % loop
     ||
     await immediate STOP ;
     await immediate Q_LOOP ;
     emit STOP_LOOP
  end % trap
end % signal
```

The LOOP construct allows iteration on a *plan.* There is no termination condition, but we use the STOP input signal seen in the declarations, in order to enable a termination decided from the external world. However, the *plan* in the loop is not interrupted: exiting the loop is possible only at the beginning of an iteration.

Therefore, two local signals are defined: Q_LOOP is emitted at each beginning of the loop, meaning that at this point the loop might be exited. If this questionning signal is answered by STOP_LOOP, then the trap statement containing the ESTEREL loop is exited. The STOP_LOOP signal is emitted only when the STOP

signal has been received and the application LOOP is ready to be terminated. This way, the LOOP construct can in fact be used in the frame of the ASSOC and UNTIL-R constructs defined further, just like the rules. Then, a conditional iteration can be written:

UNTIL-R(*condition*, LOOP(*plan*)) .

where the execution of the *plan* will be repeated until the *condition* is met. As will be seen further, in the code produced by the translation of UNTIL-R, the STOP signal is emitted when the *condition* is satisfied: then, through the STOP signal, the second part of the parallelism will wait the next occurrence of Q_LOOP asking for continuation, and answer with STOP_LOOP, which will lead to exiting the loop.

the association of a set of *tasks* to a *plan* :

< AS-LONG-AS(*plan* , *tasks*) > ⟶

```
trap END_AS_LONG_AS in
 <plan>
 ;
 exit END_AS_LONG_AS
 < tasks >
end % trap
```

where *tasks* is translated by a concatenation of, for each *task*:

```
 ||
 < task >
```

Here, in the trap construct, the exit statement, executed immediately after the termination of the code corresponding to the *plan*, forces the termination of the *tasks*.

the association of a set of *tasks* to a *condition* :

< UNTIL(*condition* , *tasks*) > ⟶

```
trap END_UNTIL in
 <waiting(condition)>
 ;
 exit END_UNTIL
 < tasks >
end % trap
```

where *tasks* is translated by a concatenation of, for each *task*:

```
 ||
 < task >
```

Like for AS-LONG-AS, the exit statement in the trap construct terminates the parallel *tasks*. Here, the termination of the waiting for the *condition* causes the termination of the rest.

a simple rule :

< WHEN(*condition* , *plan*) > ⟶

```
trap END_RULE_condition in
 do
 <waiting(condition)>
 watching immediate STOP
 timeout exit END_RULE_condition
 end % do
 ;
 < plan >
end % trap
```

As seen earlier, the execution of a rule must be interruptible, except when the reaction *plan* is executing. In order to avoid the interruption of an executing reaction *plan*, we have to use synchronization dialogues. Therefore, a signal STOP will be used, its reception being treated as a request for rule termination. It is declared elsewhere as a local signal (e.g. in the ASSOC or WHENEVER primitives) or global input signal (it can then be connected to the user interface). When received during the *waiting* for the *condition*, the signal STOP interrupts it, and the watching statement terminates by causing the exit of the trap. Otherwise, the termination of the *waiting*, corresponding to the occurrence of the *condition*, is immediately followed by the execution of the reaction *plan*.

The name of the trap: END_RULE_*condition*, is made from the *condition* name in order to avoid an ESTEREL compiler warning message in case of imbricated rules using the same trap identifiers, e.g.:

WHEN(C_1, WHEN(C_2, ...))

If the condition is TRUE, then:

< WHEN(TRUE , *plan*) > $\longrightarrow$ < *plan* >

a repetitive rule :

< WHENEVER(*condition* , *plan*) > $\longrightarrow$

```
signal STOP_WHENEVER in
 trap END_WHENEVER in
   loop
     < WHEN( condition, plan ) >
     ;
     emit STOP_WHENEVER
     ||
     do                % to avoid an immediate loop
       await STOP
     watching STOP_WHENEVER
   end % loop
   ||
   await immediate STOP;
   await immediate STOP_WHENEVER;
   exit END_WHENEVER
 end % trap
end % signal
```

The iterative rule corresponds to the simple rule WHEN in a loop. It must be interruptible, but only between executions of the reaction *plan*. Therefore, each execu-

tion of the repeated simple rule is followed by the emission of the STOP_WHENEVER signal, meaning that the WHENEVER construct might be terminated if the termination was required from outside by a STOP signal (this latter signal is used in a way similar to the case of WHEN). A branch parallel to the loop, terminating simultaneously, is there in order to avoid an otherwise possible instantaneous execution of the body of the loop, which is forbidden by ESTEREL.

The WHENEVER construct terminates by **exiting** the **trap**, when the STOP required from outside is confirmed by the STOP_WHENEVER indicating that the reaction is not being executed.

the association of a set of loops or rules to a plan :

< ASSOC(*plan* , *loops or rules*) > ⟶

```
signal STOP in
 < plan >
 ;
 emit STOP
 < loops or rules >
end % signal
```

where < *loops or rules* > is the concatenation of, for each *loop* or *rule*:

```
 ||
 < loop or rule >
```

In a way comparable to the AS-LONG-AS construct, the termination of all the *rules* is required by sending a local signal STOP immediately after the termination of the *plan* (remember that using an **exit** or a **watching** statement would cause the active reactions to be killed, which we do not want).

Once the STOP signal is emitted, only the reactions currently executing are not stopped: no condition is awaited anymore, and none of the reactions will be fired. In this way, the execution of the ASSOC construct is independent of the order of the *rules*.

the association of a set of loops or rules to a condition :

< UNTIL-R(*condition* , *loops or rules*) > ⟶

```
signal STOP in
 < waiting(condition) >
 ;
 emit STOP
 < loops or rules >
end % signal
```

where < *loops or rules* > is the concatenation of, for each *loop* or *rule*:

```
 ||
 < loop or rule >
```

As for ASSOC, the local signal STOP is sent immediately after the termination of the *waiting* for the *condition*.

a precedence statement : as said before, the processing of *precedence points* is made by instantaneous dialogues using ESTEREL local signals.

For each *precedence point* P, the following local signals: Q_P, OK_P, GO_P, THANX_P, are declared in a **signal** ... **in** statement containing all the body of the ESTEREL module. These signals are used in a dialogue between each of the parallel branches where the BLOCK and UNBLOCK instructions are used, and a synchronizing dialogue, put in parallel of the translation of the body of the application. The synchronizing dialogue is iterative: it can accept a sequence of BLOCK and UNBLOCK instructions in alternance. In order to stop this iteration, and allow the parallel construct to terminate, a **trap** statement is used, exited when the *body* is executed.

```
signal
 Q_P ,
 OK_P ,
 GO_P ,
 THANX_P
signals for the other precedence points
  ;
in
trap END_PRECED in
 [
  every OK_P do
     emit GO_P ;
     present THANX_P
        else
           await Q_P ;
           emit GO_P
     end % present    end % do
 ||
   code for the other precedence dialogues
 ||
   code for the body of the application
   ;
   exit END_PRECED  ]
end % trap
end % signal
```

This part of the dialogue awaits a releasing signal OK_P from the UNBLOCK instruction, before warning a possibly blocked BLOCK instruction with a GO_P signal. If the BLOCK instruction is executing and blocked, then it answers with a THANX_P, else it later emits Q_P: then the dialogue re-emits GO_P.

Thus, the code for the two instructions is:

blocking on a precedence point P :

< BLOCK(P) > $\longrightarrow$

```
emit Q_P ;
await immediate GO_P;
emit THANX_P ;
```

This code first emits a question Q_P, meaning "can I pass?", and awaits a (possibly immediate, if the unblocking is already done) answer GO_P from the first part of the dialogue, meaning "you can go". Finally, it emits a signal THANX_P so that the handler knows about it.

unblocking on a precedence point P :

< UNBLOCK(P) > $\longrightarrow$ `emit OK_`P

This code simply releases the blocking in the first part of the dialogue.

The fact that local signals are used for the synchronization makes the use of precedence points costless in terms of the automaton produced by the ESTEREL compiler. This means that using a SEQ construct, or programming the sequence with precedence points produces the same automaton. In the example shown previously in figure 9, the application specifying these precedence relationships could be:

```
PRECED PA.
PRECED PB.
PRECED PC.
PRECED PD.
PRECED PE.
PRECED PF.
PRECED PG.

PAR(
       SEQ( P1, UNBLOCK(PA), UNBLOCK(PB) ),
       SEQ( BLOCK(PA), P2, UNBLOCK(PC), UNBLOCK(PD) ),
       SEQ( BLOCK(PB), P3, UNBLOCK(PE) ),
       SEQ( BLOCK(PC), P4, UNBLOCK(PF) ),
       SEQ( BLOCK(PD),BLOCK(PE), P5, UNBLOCK(PG) ),
       SEQ( BLOCK(PF),BLOCK(PG), P6 )
).

END.
```

where each arrow, representing a precedence, is implemented by a a precedence dialogue.

The advantage of this is obviously not the improvement of programming style. However, it is interesting in the perspective of having to translate such precedence graphs, generated automatically by an A.I. planner, into executable ESTEREL programs: then, even without possibly using the constructs of the applications programming language, the translation seems cost-effective, in terms of the generated code.

3.6 A tentative language extension for error processing

This section describes propositions that have not yet been implemented and tested with the remaining of the language. They are nevertheless developped in a way that is coherent with what was presented above.

3.6.1 The error processing functionality

Upto now, only few attention was paid in the language and its translation into ESTEREL, to the detection and processing of errors and exceptions: in fact, the only reaction at application level was to stop everything and call an operator for help. However, as already evoked in section 2.2, the concept of multi-level reaction to execution failures already leads us to distinguish between three different behaviours, according to the interruption level:

at task-level : the error is treated inside the task, and is invisible and inoffensive externally (processings 0 and 1, in section 2.2.6) ;

at a sub-plan level : the effect of the error is restricted to a given sub-plan, that must be delimited, and is associated with a processing, that is another sub-plan (possibly `NOTHING`)(processings 2, in section 2.2.6) ;

at global level where the whole application is stopped, and the control is given to the operator (processings 3, in section 2.2.6) .

In the framework of our application programming language, we want to introduce these concepts by defining error processing modes for plans and sub-plans in the applications, associating different behaviours to different parts, but with a clear overall control structure.

A default mode for the applications (when nothing particular is specified) is the task-level processing behaviour, where the robot task itself handles the occured error and no signal is emitted towards higher levels in the plan. Then, execution proceeds in the "sequencing" of actions as if the concerned task had normally terminated. This behaviour corresponds to an execution at the user's own risk, with no particular error handling, i.e. it corresponds to the first level.

Another mode can be characterized by the propagation of the error to higher levels in the application: this propagation can be limited, "filtered", at some level of the application (e.g. one branch of a parallel construct), and a processing can be associated to it, which corresponds to the sub-plan level. An unlimited propagation, upto the application's top-level and the user, correspond to the global level: the related processing consists in stopping the whole application.

We will here propose an extension to the application programming language enable to handle these error-dedicated processings, by giving in the following new control structures therefore, examples illustrating their properties, and the scheme of their translation into ESTEREL.

3.6.2 The new language features

Addenda to the grammar. Concerning the syntax, the following rules must be added to the language grammar, introducing the three control structures that will take error handling specification in charge:

plan ::= ...
| WEAK-ERROR(*plan*)
| STRONG-ERROR(*plan* { , *plan* })
| FATAL-ERROR(*plan*)

Addendum to the robot-tasks. In the basic features of the language, the robot tasks are modified in order to take errors into account. Indeed, until now there was no emission of error signals: there were just observations, for which the processing was done internally to the task. We are now interested in how to detect the errors and to propagate them to upper levels, regardless of what these upper levels will do with them. To this end, we will now define a robot task by:

- preconditions, observations and their associated processings, postconditions, and a control procedure, as before;
- a set of *error conditions*: for each of them, each time it is true in the environment, an error signal is emitted to the upper level, and the robot-task is interrupted and aborted.

Error processing control structures.

the "weak error" processing :

WEAK-ERROR(*plan*)

The weakest processing to react to errors consists in not doing anything, at any level. In this sense, any error signal emitted inside the *plan* will be ignored, "absorbed', and its propagation will be stopped : it will be invisible outside the WEAK-ERROR construct, and have no influence.

What happens in a more detailed way, when an error occurs in the *plan*, is that the robot task, from where the error signal was issued, is aborted, and the execution of the *plan* continues "in sequence", i.e. following the control structure as it was specified. The user who writes the application must take this fact into account, and be sure that his plans are robust enough to be executed with some errors.

This error processing is the default mode when executing the top-level application : when specifying nothing else, this is the behaviour that it will have. However, if one wishes that the whole application behaves in one of the two other modes, it is possible to englobe the whole body of the application in one of the corresponding two other control structures.

the "strong error" processing :

STRONG-ERROR(*plan* , *processing-plan*)

The strong error processing is applied to the first *plan*, and consists, in case of strong error occurrence in it, in aborting it, and executing immediately in sequence the *processing-plan* (that may be NOTHING), which then terminates the STRONG-ERROR construct. If no error occurs, then only the *plan* is executed, completely.

The error is treated at the level where it is received : i.e. the fact that it arrived upto there means that it was not "absorbed" lower, and as it is treated there, it will not be propagated above, and will have no influence outside the scope of STRONG-ERROR.

the "fatal error" processing :

FATAL-ERROR(*plan*)

The last error processing, treating cases where there is nothing to be done, i.e. "fatal" cases, consists in transforming an error emitted in the *plan*, into a global error signal that will not be absorbed, i.e. it will be propagated upto the top-level, and interrupting and aborting the global application. It corresponds in fact to the "big red button" on machine-tools.

Another way of presenting these three error processing control structures, or modes, is to see the two first, WEAK-ERROR and STRONG-ERROR, as filters stopping the propagation of error signal coming from inside their scope towards the top-level of the application. WEAK-ERROR receives them and stops them, without treating them: it is as an opaque filter. STRONG-ERROR receives and stops them, and treats their occurrence by abortion of the *plan* and execution of the *processing-plan*. The errors occurring inside these two constructs are treated as relative errors. In contrast to this, FATAL-ERROR receives and stops error signals from inside its scope, and transforms them into another special signal propagated upto the top-level of the application, without being stopped. In this sense, it transforms errors occurring inside its scope into absolute errors.

3.6.3 Examples

In order to clarify these behaviours and their interactions, we will give some examples.

- In a sequence, in case of error occurrence in the sub-plan P_i, the behaviour can consist in:
  - aborting its execution, executing an associated processing plan T_i, and proceeding in sequence to P_{i+1}, as in:
    SEQ(P_1, ... , STRONG-ERROR(P_i,T_i) , P_{i+1}, ...)
  - aborting the whole application, aborting P_i and withdrawing all sunsequent sub-plans $P_{j,j>i}$:
    SEQ(P_1, ... , FATAL-ERROR(P_i) , P_{i+1}, ...)
- Inside a FATAL-ERROR or a STRONG-ERROR construct, it is possible to "filter" error signals coming from some parts of the plan. For example, in:

```
STRONG-ERROR(  PAR(  B1 ,
                     B2 ,
                     WEAK-ERROR( B3 ) ,
                     B4 ) ,
               P_processing )
```

  if an error occurs in B_1, B_2 or B_4, then $P_{processing}$ is executed after all the branches are aborted; but errors from B_3 are not taken into account.

In another example:

FATAL-ERROR(..., WEAK-ERROR(P) , ...)

an error occurring in P is "filtered out" by WEAK-ERROR and does not entail a fatal error at the top-level.

- In a parallel construct, the different branches can be provided with different behaviours. For example in the application:

```
STRONG-ERROR( PAR( B1 ,
                   STRONG-ERROR( B2 , T2 ) ,
                   WEAK-ERROR( SEQ( P1 ,
                                    FATAL-ERROR(P2),
                                    P3 ) ) ,
                   B4 ) ,
              Pprocessing )
```

if an error occurs in:

  - B_1 or B_4: then the plan $P_{processing}$ is executed, after everything has been aborted in the PAR statement.
  - B_2: then B_2 is aborted and the processing T_2 is executed; it is a local error processing: the other branches in the PAR statement proceed undisturbed, the error is not propagated to the above STRONG-ERROR statement.
  - P_1 or P_3: only the task from where the error comes is aborted, the sub-plan containing it is continued in sequence, the error is not propagated to the above STRONG-ERROR statement.
  - P_2: the whole application is aborted: the fatal error signal is propagated upto the top-level, it "breaks through" the WEAK-ERROR and STRONG-ERROR constructs.

3.6.4 Translation into ESTEREL

The translation scheme described in 3.5 has to be adapted in order to accept these extensions.

Top-level module. Until now, there were no output signals; but the fatal error processing requires to emit a warning signal to the external world, namely to the operator. A declaration must therefore be made (between the input and task declarations):

```
output
 ERROR
 ;
```

Since the default error processing mode is WEAK-ERROR, at the top-level the body of the application will be placed inside a local signal declaration, that will "filter" the signal emissions. In order to treat the FATAL-ERRORs, a trap statement will be "exited", with a handle statement consisting in emitting the output signal ERROR, outside the local signal declaration.

Thus, for an application having a given *body*:

```
trap FATAL in
 signal ERROR in
    < body of the application >
 end % signal
handle FATAL do emit ERROR
end % trap
```

Robot tasks. In the robot tasks, we have to produce the code corresponding to the behaviour associated to this particular type of observer: observation of the error conditions, emission of an error signal when the condition is verified and exiting of the task ; this code runs in parallel with the execution of the control, the observations, and the postconditions, as seen in section 3.5:

```
trap END-TASK in
 < command >
 ||
 < observations >

 < error condtitions >
 ||
 < postconditions >
end % trap
```

where *error conditions* consist in the concatenation, for each of them C, of:

```
 ||
 < waiting(C) > ;
 emit ERROR;
 exit END_TASK
```

If an error condition C is met, then the error signal is emitted, in order for the upper levels to activate the according processings (weak, strong or fatal), and the task is aborted, for it is the only thing it can do in case of error.

Control structures.

Weak error processing :

<WEAK-ERROR(*plan*)> $\longrightarrow$

```
signal ERROR in
 < plan >
end % signal
```

The local signal declaration of ERROR makes that if it is emitted in the *plan*, then it will not be received outside the local signal declaration, because it is not defined there. In this way, it is "filtered out".

Strong error processing :

<STRONG-ERROR(*plan* , *processing*)> $\longrightarrow$

```
trap T_ERROR , OK in
 signal ERROR in
   < plan > ;
   exit OK
   ||
   await ERROR ;
   exit T_ERROR
 end % signal
handle T_ERROR do
 < processing >
end % trap
```

The local signal ERROR is awaited, and when received, the trap T_ERROR is exited, causing the abortion of the execution of the *plan*, and, in the trap **handle** statement, the *processing* plan is executed. This way, the ERROR is treated at this level, and not propagated (i.e. emitted) further.

If there is no error occurrence, then the plan is executed normally, and the trap is exited through OK.

Fatal error processing :

<FATAL-ERROR(*plan*)> $\longrightarrow$

```
trap T_ERROR , OK in
 signal ERROR in
   < plan > ;
   exit OK
   ||
   await ERROR ;
   exit T_ERROR
 end % signal
handle T_ERROR do
 exit FATAL end % trap
```

If an error signal is received, then the trap is exited through T_ERROR, the plan is aborted and in the **handle** statement, the top-level trap FATAL is exited, aborting the whole application.

Otherwise, the *plan* is executed normally, and the construct exited through OK.

4 Applications

4.1 Interfacing the asynchronous environment to the synchronous automaton

Introduction At this step, we are given an application programming language and a methodology of robot-task design.

Once a robotic application has been defined, in terms of a sequence of robot actions, the last step consists in executing the associated programs. They are considered as *"reactive programs"* because of the reactions induced by all the different input events. These reactions consist in generating output events toward the environment. These programs are always interacting with the surrounding environment, made of many process evoluating in *parallel* and *asynchronously* (including human operator). In our case, the synchronous language ESTEREL is used in the way previously presented to specify the appropiate behaviour of the system. It allows to describe the logical synchronization skeleton in charged with the control of several asynchronous activities. These last parts are generally programmed using a General Purpose Language (GPL as stated in [Berry89]).

Therefore an interface between the skeleton and the asynchronous tasks is required. This interface is called an *"Execution machine of* ESTEREL *programs"*. We briefly describe an abstract execution machine[1] and give the general architecture of such a machine. A definition and the role played by the various parts are mentioned. No reference is made to any operating system, and the reader will find a complete description of this machine in [André e.a.91]. An example of a particular one based on the UNIX system dedicated to robotics application can be found in [Coste-Manière e.a.90] and in section 4.3.2.

Recalls Let us consider a given application, the logical aspects of which have been expressed using the ESTEREL language. We recall that ESTEREL is based on the following paradigm[2]: A program acts as a *synchronous history transformer* that produces a sequence of outputs *synchronously* with any sequence of inputs. The synchrony hypothesis is of course an approximation of the reality and it ideally assumes that a synchronous program behaves as if its internal activities were carried out by an infinitely fast machine. In reality, one has to carefully evaluate the application range of the synchrony hypothesis and to define a "satisfaction point" where a user perceives the system as synchronous: the reaction to an input can take time but must be uninterruptible, that is to say no new input can be taken into account during the execution of the reaction. Therefore an implementation satisfies this hypothesis as long as the reaction time is always short enough to avoid overruns ("reaction time" means the duration of sequences of actions associated to a *reaction*). To avoid overruns, the execution of the small and efficient automaton, result of the program compilation, must be controlled. This is performed in three steps:

First of all, before executing a reaction, i.e. before activating the automaton, it is necessary to "take a picture" of the environment, and to build a *current event*: all

[1] The scheme of an execution machine given here is not limited to robotics applications and is required in the implementation of all kinds of reactive applications

[2] This paradigm is common to all the synchronous formalisms

the signals present on the picture are said to be simultaneous. In other words, they were emitted in the "same instant". Then the automaton must be activated through an extra signal call. All the actions associated with the unique possible transition (this uniqueness is linked to the determinism: one image, one state, one transition) are finally executed and the next automaton's state is reached, once the transition is over.

In the following, we provide the ESTEREL programmer with all the facilities required to take the picture of the environmnent and to perform the various actions (see figure 10 taken from [André e.a.91]).

Input/Output Processing: Input processing is dedicated to external signal collecting, to information extraction and storage, in order to define the current event.

Output processing is required in order to transmit to the environment the result of the actions execution.

Both of them act as classical input/output drivers, and the policies to detect new signals occurences (input driver) are polling or interruption handling.

The Reactive Machine As mentionned above, synchrony is an approximation of the reality, and the automaton has to deal with an *asynchronous world.* In order to manage the information updated asynchronously, a so called *reactive machine* was designed. It is made of three sub-machines: an *event generator*, the *automaton* itself and a *synchronous action processing machine.*

The *event generator* (or "history maker") uses the information stored by the input processing: it is dedicated to current events making. The history can be created according to various strategies such as self-synchronisation (an unique signal in the current input signal) or a clock driven policy (the automaton is activated at every clock tick, and signals are grouped in between two successive clicks).

Once the current signal is built the reaction can be achieved (through the call of the activation signal): starting from a current state, the *automaton* can compute the next one and control the execution of the reactions. The actions associated to a transition are controlled by using a *synchronous action processing.* This sub-machine cooperates with the *asynchronous machine* especially when an **exec** statement is required.

The asynchronous machine This part is necessary to preserve safety and determinism brought by the synchronous language, while executing asynchronous actions, that is to say actions that last more than a reaction. These actions are performed by using an external task activated by the **exec** statement (see [Paris91] for a detailed presentation). As mentioned in section 2.2.6, whenever an **exec** is encountered, the compiler generates automatically three signals: `Start`, `Terminated`, `Kill`. The **exec** *manager* sub-part receives signals coming from the synchronous part and dispatches the corresponding orders (`Start` or `Kill`) to the asynchronous part in charge of tasks computation. It acknowledges receipt of tasks termination to the *Information storage.*

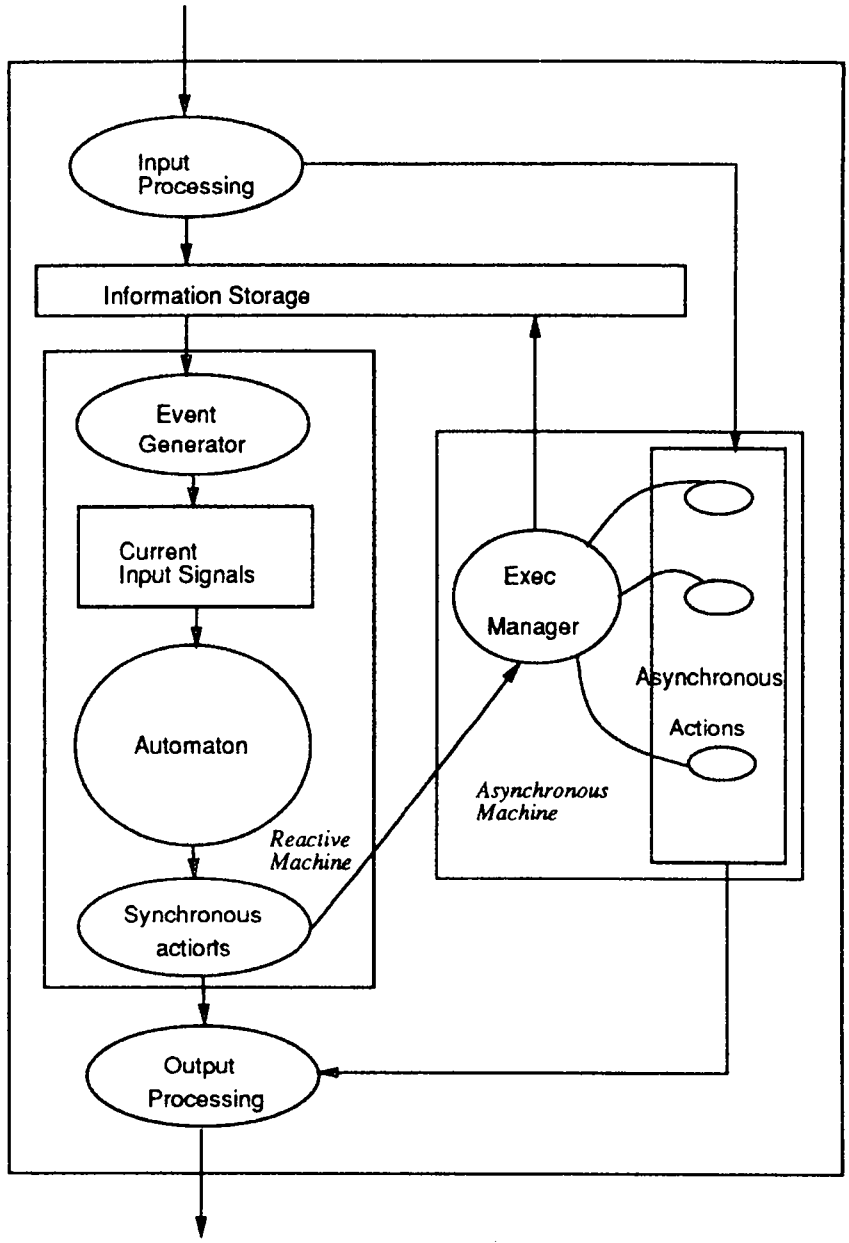


Figure 10: General configuration of an ESTEREL executing machine

4.2 A Simple Example

4.2.1 Script of a simple assembly application

We now illustrate the overall approach (i.e. Robot-tasks design, sequencing, simulation) through three examples, developped in this section and the two next ones.

We consider a single assembly cell within a flexible manufacturing system. The objective to be realized consists in manipulating a part according to the following script: A 6 degrees-of freedom robot, `RobotA`, grasps an object on a conveyor belt `BeltA`, and poses it on the upper platform of a force-controlled parallel manipulator [Merlet91], called a `LeftHand`. Then, a second 6 d.o.f. robot, `RobotB`, grasps the part on the `LeftHand` and then moves to an insertion place where the grasped object is inserted. Several signals, issued from sensors or emitted by the control system, rythm the application sequencing. Furthermore, in the framework of a continuous flow, a minimization of the average processing time in the cell is wished, which implies to operate in *parallel* as far as possible.

4.2.2 The Application Program

4.2.3 The assembly cell

The different actions to be executed on this assembly cell are:

- for the `BeltA`: each time its extremity is free, to make the belt move ahead in order to bring on a new object.
- for the `RobotA`: each time there is an object on the `BeltA`: to grip it ; then to move to the `LeftHand` ; then, when the `LeftHand` is empty, to put the object on it ; then to move to the `BeltA`.
- for the `RobotB`: eachtime the `LeftHand` is loaded (i.e. not empty): to move to the `LeftHand`, then to grip the object on the `LeftHand` ; then to move to an insertion place ; then to insert the object in this place.

In the robot system model, we have:

- conditions:
  - `BeltAFree` is true when there is no object on the extremity of the conveyor: the sensor giving this information is a photoelectric cell ;
  - `LHempty` is true when there is no object in the `LeftHand`: the sensor giving this information is a weight sensor `Weight_sensor`.
- robot tasks:
  - movement of `RobotA` to `BeltA`: `RAmoveToBeltA`,
  - gripping of an object by `RobotA` on `BeltA`: `RAgripOnBeltA`,
  - movement of `RobotA` to `LeftHand`: `RAmoveToLH`,
  - putting an object by `RobotA` on the `LeftHand`: `RAputOnLH`.
  - movement of `RobotB` to `BeltA`: `RBmoveToLH`,

- gripping of an object by RobotB on LeftHand: RBgripOnLH,
- movement of RobotB to InsertPlace: RBmoveToInsertPlace,
- insertion of an object in the InsertPlace: RBinsertInPlace,
- control of the conveyor (movement of BeltA): BeltAcontrol.

4.2.4 Some Related Object Instances

We give in this section few examples of objects to be created for fully specifying this application. Only some relevant ones, concerned with robot-tasks are presented. Ideally, the mainly required objects would be identified as follows:

Robot-tasks

RT.RA.Motion1 (RAmoveToBeltA)
RT.RA.Motion2 (RAmoveToLH)
RT.RB.Motion1 (RBmoveToLH)
RT.RB.Motion2 (RBmoveToInsertPlace)
RT.RA.Grasping (RAgripOnBeltA)
RT.RB.Grasping (RBgripOnLH)
RT.RA.Ungrasping (RAputOnLH)
RT.RB.Insertion (RBinsertInPlace)

Task functions

TF.TT.SE3.RA.Motion1
TF.TT.SE3.RA.Motion2
TF.TT.SE3.RB.Motion1
TF.TT.SE3.RB.Motion2
TF.RT.WS.RA.Grasping
TF.RT.WS.RB.Grasping
TF.RT.WS.RB.Insertion

Remarks: There is no need for an *ungrasping* task function since it simply consists in opening the gripper, while *grasping* includes a proximity-sensor aided motion. Recall that other required objects for building robot-tasks belong to *observers*, *trajectory generators*, *controls* and *models* classes.

Resources

RE.CB.BELTA
RE.ROB.LH
RE.ROB.RA
RE.ROB.RB
RE.SE.FI.RO.RA.Prox

RE.SE.FI.RO.RB.Prox

RE.SE.FI.RO.LH.Force

Let us now give further caracteristics of some selected objects. They are indeed presented under a "naive" form, since no actual implementation has yet be realized.

Instanciation of an Object of the Class *Task Function*

Class TF

Attributes

robot = *RA*

dimension = *6*

time_duration = *3 sec*

$\lambda(t) = 0$

initial_condition = *measured*

Methods

symbolic computation: cancelled

coherence checking: cancelled

Continuation: trajectory tracking ; in SE_3

Object TF.TT.SE3.RA.Motion1

Attributes

concerned_frame = (default) = *GRIP_FRAME*

parametrization = u, θ

tracked_trajectory $x_r(t)$: GT.SE3.POL.RA.Motion1

used_kinematics $x(q)$: MOD.KIN.DIR.RA.GRIP_FRAME

attitude_error_type $E_a(q,t)$

Methods

$$\textit{computation of} \begin{pmatrix} x(q) - x_r(t) \\ E_a(q,t) \end{pmatrix}$$

Instanciation of Objects of the Class *Models*

Class MOD

Attributes

robot = *RA*

Continuations: Dynamics ; Kinetics Energy ; other terms

Object MOD.DYN.KE.RA.Simple

Attributes

model_choice = *constant*

measurements = {*none*}

parameters =...

Methods

$\hat{M}$ = *parameters*

Object MOD.DYN.ETC.RA.Simple

Attributes

model_choices:

gravity = *computed*

coriolis/centrifugal = *0*

friction_model = *0*

measurements = $\{q_1 ... q_6\}$

Methods

computation of $G(q)$

Object MOD.KIN.DIR.RA.GRIP_FRAME

Attributes

measurements = $\{q_1 ... q_6\}$

output = M *(4×4 homogeneous matrix including the position and the rotation matrix)*

end_frame = *GRIP_FRAME*

Methods

computation of $M(q)$

Object MOD.KIN.INV.RA.GRIP_FRAME

Attributes

end_frame = *GRIP_FRAME*

input = M

solution_choice = *singularities_removal*

Methods

computation of $q(M)$

Instanciation of an Object of the Class *Robot-tasks*

Object RT.RA.Motion1

Attributes

task_function selection: TF.TT.SE3.RA.Motion1

precondition 1: RA_in_the_neighbourhood_of_BeltA = *false*

precondition 2: gripper_empty = *true*

observer selection: OBS.TF.error (size of the tracking error e)

threshold =σ

postcondition:RA_in_the_neighbourhood_of_BeltA = *true*

Methods

ESTEREL *encoding of emergency stop when* $||e|| - \sigma > 0$

ESTEREL *encoding of pre- and post-conditions handling*

4.3 The Application

Using these *conditions* and robot *tasks*, we can write the application in the applications programming language: it begins with the declaration of one *compound task* for each of the components of the assembly cell, specifying the behaviour described before. Then, they are used in a parallel construct to specify the *plan*.

```
PLAN BeltA :    SEQ(   WHENEVER( BeltAFree, BeltAcontrol) ) .

PLAN RobotA :   SEQ(  WHENEVER( NOT BeltAFree,
                                SEQ(    RAgripOnBeltA,
                                        RAmoveToLH,
                                        WHEN( LHempty,
                                              RAputOnLH),
                                        RAmoveToBeltA ) ) ) .

PLAN RobotB :   SEQ(  WHENEVER( NOT LHempty,
                                SEQ(    RBmoveToLH,
 RBgripOnLH,
                                        RBmoveToInsertPlace,
                                        RBinsertInPlace ) ) ) .
PAR(    BeltA,
        RobotA,
        RobotB ).
END.
```

In the case of this application, the structure of the plan allows the use of the CASCADE form, that enables the use of an optimisation option of the ESTEREL compiler.

Remark: A direct ESTEREL implementation of the same example is given in Annexe B. Nevertheless, it includes some kind of optimization (parallelism, synchronization points ...). We also encode the logical and temporal dependences, between robot-tasks leading to their global scheduling. This is performed using rendez-vous, semaphores, instantaneous dialogues... At the last step, all the ESTEREL elementary programs (intra- and inter robot-tasks) have to be concatenated in order to generate a single global automaton, simulations or validation of which are allowed using for example AUTO or AUTOGRAPH tools (see [deSimone e.a.89], [Roy e.a.89]).

4.3.1 Proofs

We now give an example of the type of proofs that can be performed, on a compiled ESTEREL program. We thus illustrate the point 1 stated in section 1 in order to satisfy the classical concern of software quality: the produced code is reliable because its execution is deterministic. This determinism is linked to the ESTEREL feature of synchronism. The translation of an ESTEREL program into a finite determinist automaton authorizes to perform automatic proofs.

In this section, we use the verification system for parallel and communicating processes AUTO [deSimone e.a.89] to prove that the behaviour of the reactive application is the expected one. Although proofs were performed on all the described applications,

we give a single example. It has been carried out on the implementation code given in appendix B.

From the specification of the application mentionned above, simple and precise properties of the application can be stated in natural language. For example: it is mentionned that RobotB goes and picks up an object on LeftHand. We want to make sure that RobotB can pick up the object if and only if an object is on LeftHand that is to say if and only if RobotA has put an object on LeftHand.

The automaton translation assumes that this property can be proved. Let us consider the signal associated to the following tasks:

OBJECT_IS_IN_LEFT_HAND	for the end of	RAputOnLH
GRIPPING_THE_ASSEMBLY	for the beginning of	RBgripOnLH

Using AUTO we observed the global automaton while focusing on these two signals. Indeed AUTO uses experiment verification principles such as the quotient of the automaton by an observational criterion. It is performed by using the following sequence:

```
% set VAR = obs (ACTION,{OBJECT_IS_IN_LEFT_HAND, GRIPPING_THE_ASSEMBLY});
VAR : Automaton

% display VAR short;
size = 3 states, 4 transitions, 4 actions.
```

The result of this observation is a restricted automaton (three states) that can be seen on figure 11. This drawing has been performed using AUTOGRAPH [Roy e.a.89].

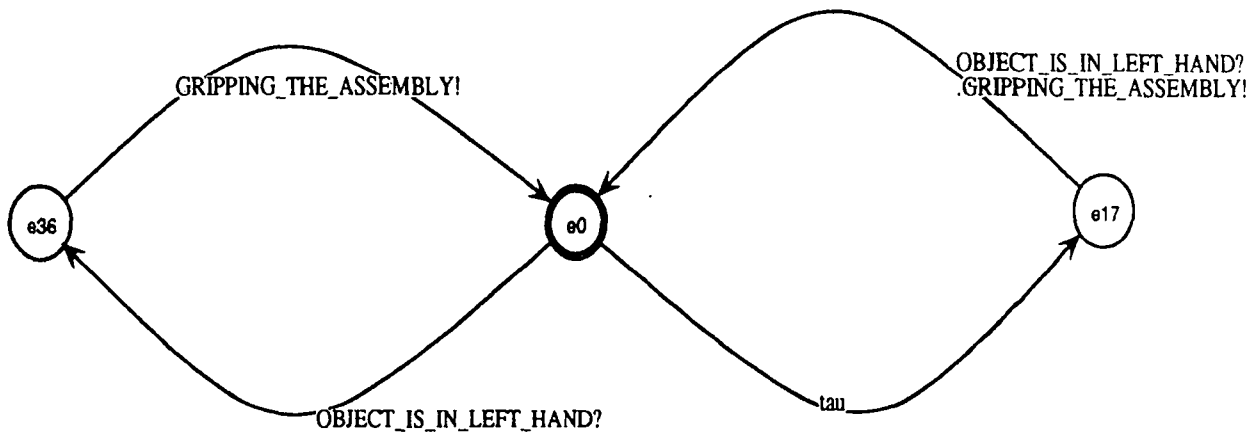


Figure 11: A restricted automaton.

We can check out that the only possible sequence of signal is:

```
--BELT_MOTION!.R1_TRAJ_TOWARD_B!.R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND!-->e1
--R2_NEAR_LH? --> 2 e2
--VICINITY_OF_B? --> 5 e5
--GRIPPING_ON_BELT!.OBJECT_VICINITY_R1B? --> 8 e8
--BELT_MOTION!.OBJECT_IN_GRIP?.TRAJECTORY_TRACKING_TOWARD_LEFT_HAND!--> e10
--VICINITY_OF_B? --> 13 e13
--LEFT_HAND_VICINITY?.PUTTING_THE_OBJECT_IN_LEFT_HAND! --> 17 e17
```

This confirms the above mentionned property: `OBJECT_IS_IN_LEFT_HAND` is always received by the system before `GRIPPING_THE_ASSEMBLY` is emitted. Therefore the task `RAputOnLH` is always ended before the task `RBgripOnLH` is activated.

4.3.2 Simulation

Some experimentation of the proposed approach have been performed on a UNIX-based graphic workstation, the assembly cell being simulated by a CAD software called RHOMEO [Faverjon89]. The execution machine presented in section 4.1 is splitted up into three UNIX processes: one is dedicated to input/output processing, the second one to the implementation of the automaton and of the event generator ; the last one handles the task management.

We only discuss here the asynchronous task manager structure, which takes advantages of the functionnalities provided by RHOMEO [Faverjon86].

RHOMEO enables the object level programming of industrial robots. It includes the following modules :

- A modeler used to build CAD models of robots and their environment,
- An obstacle avoidance module, that is able to compute collision free trajectories using various methods,
- A planning module that translates object level instructions into robot level instructions with the help of the previous module.
- A graphical simulation module which simulates the computed trajectories.

RHOMEO uses hierarchical models of solids and articulated chains that enables efficient collision and distance computations. See [Faverjon89] for more details.

Implementation Using RHOMEO we have realized a tool allowing a graphical animation of the application, i.e. a simulation of a real control and programming system, dealing with a variety of robotics applications. The design of an application simulation is done as follows.

- The preliminary step consists in creating the different robots involved in the robotics cell. This is performed using RHOMEO modeller.

- Once the cell geometry is designed, robot trajectories are to be computed. The module implements various methods, including algorithms of collision avoidance. This step corresponds partially to the robot-tasks instantiations.

- Then, according to automaton's requests (output signals generated through the signals emittor), a process maintains a table of active tasks. This allows to determine when a distributed activity is completed or to abort the corresponding task when necessary. This process is also in charge of acknowledging the end of tasks by sending input signals to the automaton.

A periodic display of the cell shows the evolution of the assembly process and simulates the computed trajectories according to a predefined assembled process.

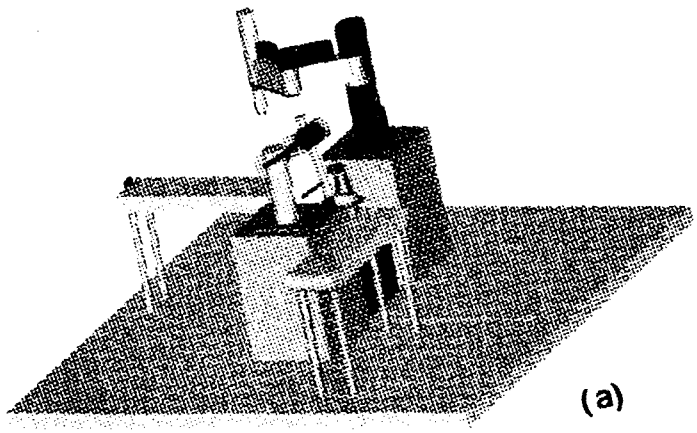
During the simulation, the user may at any time, trigger an error signal to check out the recovery scheme through out a man-machine interface.

Using this execution machine, we have simulated the application above described as well as a more complex one presented in the next section. The following figures illustrate the simulation phase :

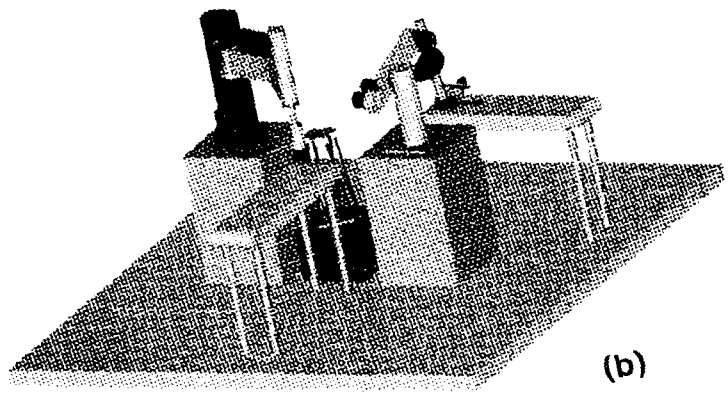
- Figure (a) shows the assembly cell design. No task is performed ;

- Figure (b) `RobotA` performs the robot-task `RT.RA.Grasping` (`RAgripOnBeltA`) while `RobotB` performs the task `RT.RB.Insertion` (`RBinsertInPlace`) ;

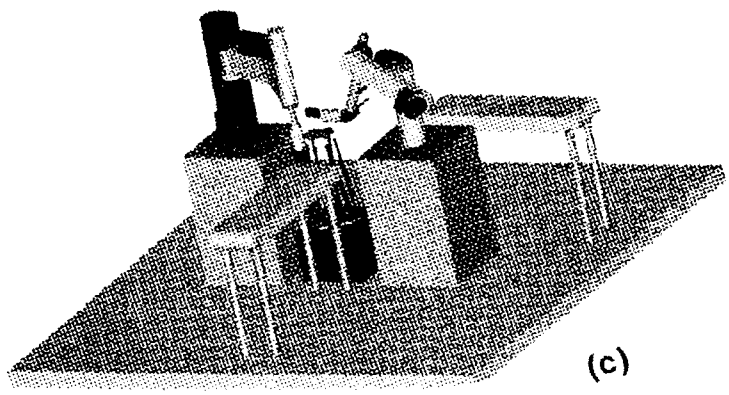
- On figure (c) `RobotA` starts the robot-task `RT.RA.Motion2` (`RAmoveToLH`) and `RobotB` has finished the task `RT.RB.Grasping` (`RBgripOnLH`).



(a)



(b)



(c)

4.4 An Assembly Example

4.4.1 Script

The following figure illustrates a more representative robot application, extension of the previous one. On this figure one can see the design of the complete assembly cell.

The tasks involve two robots cooperating in an assembly operation. Parts are fed by two conveyor belts. The robot system has to achieve the following tasks.

RobotA grasps the first part on conveyor belt BeltA using a specified gripping force program. This involves sensor-based operation to detect the arrival of an object on the conveyor. Then it places the part on the force-controlled parallel manipulator, (LeftHand) where it is locked. RobotA grasps the second part on the second conveyor (BeltB), and puts both parts together through cooperative control between LeftHand and RobotA. RobotB grasps the assembled parts, once the assembly is unlocked, and puts it inside a storage unit.

Note the central role played by sensory information: the assembly involves sensor based operations which require sensors to determine the position of the manipulators at any instant of time, vision sensors, touch sensors (for gripping)...

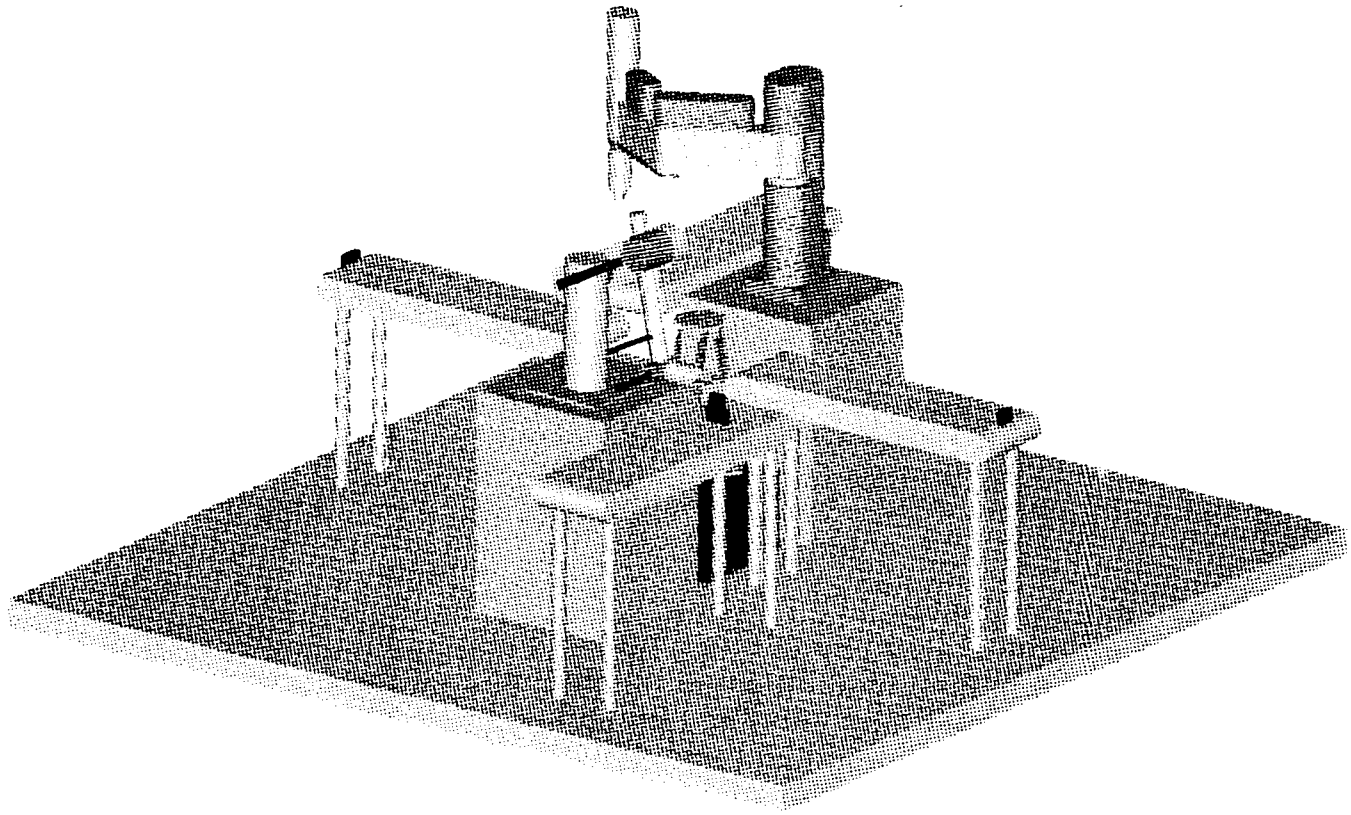
This application requires also parallel operations. For example, while RobotB is picking up the assembly, RobotA fetches another part and proceeds with the insertion. This cycle is repeated until all the parts are inserted. It illustrates particularly the advantages linked to the use of the ESTEREL language.The complete code can be found in [Coste-Manière89].

4.4.2 The Application Program

The related plan is:

- for the BeltA: the same as in section 4.2.3 ;
- for the BeltB: the same as for BeltA ;
- for the LeftHand: in a loop, once initialized, when loaded, to lock on the load, then, when the other part is in place for insertion, to cooperate with RobotA to the assembly , and then to unlock ;
- for the RobotA: once initialized, in a loop, to move to the BeltA, then to grip the object on it, then to move to the LeftHand, then, when it is empty, to put the object on it, then to move to the BeltB, to grip an object on it, to move to the LeftHand, and when in place for insertion, to participate to the assembly (with the LeftHand) ;
- for the RobotB: once initialized, in a loop, to move to the LeftHand, when it is loaded, grip the assembly on it, then to move to an insertion place, and to insert the assembly there.

The conditions used for synchronization are:



- **BeltAFree** and **BeltBFree**, acquired through photo-electric sensors,
- **LHempty**, which is acquired through a weight sensor on the **LeftHand**,
- **InPlaceForInsertion**, meaning that both objects to be assembled are present for the assembly to begin, acquired through a photo-electric sensor, or maybe *some more precise device*.

```
% compound tasks declaration %

PLAN BeltA :     SEQ(   BeltAinit,
                        WHENEVER( BeltAFree, BeltAcontrol) ) .
PLAN BeltB :     SEQ(   BeltBinit,
                        WHENEVER( BeltBFree, BeltBcontrol) ) .
PLAN LeftHand : LOOP( SEQ(   LHinit,
                             WHEN( NOT LHempty, LHlocking),
                             WHEN( InPlaceForInsertion,
                                   LHinserting),
                             LHunlocking ) ) .
PLAN RobotA :    SEQ(   RAinit,
                        LOOP(   SEQ(   RAmoveToBeltA,
                                       RAgripOnBeltA,
                                       RAmoveToLH,
                                       WHEN( LHempty, RAputOnLH ),
```

```
                                        RAmoveToBeltB,
                                        RAgripOnBeltB,
                                        RAmoveToLH,
                                        WHEN( InPlaceForInsertion,
                                              RAinsertInLH) ) ) ) .

PLAN RobotB :   SEQ(   RBinit,
                       LOOP(   SEQ(  RBmoveToLH,
                                     WHEN( NOT LHempty,
                                           RBgripOnLH),
                                     RBmoveToInsertPlace,
                                     RBinsertInPlace ) ) ) .

% application %

CASCADE(    BeltA,
            BeltB,
            RobotA,
            RobotB,
            LeftHand ) .
END.
```

4.4.3 Particular aspects of applications programming

The advantages of using the CASCADE construct. The CASCADE construct is used here because, as in section 4.2.3, the parallel branches, each corresponding to a different actuator, do not communicate through internal programmed protocols, but through external information acquired by the sensors. It is thus possible to use ESTEREL's -cascade option.

In the case of the simple example of section 4.2.3, using the PAR construct leads to ESTEREL producing a 2651 states automaton, and a 13110823 bytes *C* code, whereas the use of CASCADE and the ESTEREL compiler's -cascade option leads to the production of 3 automata respectively 6, 54 and 11 states (i.e. a total of 71 states), and a 38871 bytes *C* code.

Coordinated robot-tasks. The application example presented here was originally described by an ESTEREL program given in [Coste-Manière89]. The assembly is made by RobotA and the LeftHand, executing parallelly the tasks LHinserting and RAinsert-InLH, synchronized by the InPlaceForInsertion condition. The PAR construct is used mainly for independent tasks ; here, the two considered tasks are executed in parallel, although they are not independent. The RAinsertInLH task will be defined with an observation signal meaning that a strong effort is measured at the end of the robot-arm, and the corresponding processing will modify the movement (by stopping it, going backwards and starting again for example). The LHinserting task will have such an observation too. The parallel execution of both can then achieve the assembly.

However, in the case of coordinated movements of different devices, which need synchronization at lower levels, like in the assembly or carrying objects with two arms, it

is better from the point of view of automatic control to consider one single task to which several actuators contribute. This entails a different programming style at application programming level: the robots are not programmed with separate compound tasks any more, but in a more complex way: in the case of this assembly, we would have an application of the form:

```
SEQ(    ...
        PAR(  sub-plan for RobotA ,
              sub-plan for LeftHand ),
        CoordinatedInsertion ,
        PAR(  sub-plan for RobotA ,
              sub-plan for LeftHand ),
        ...)
```

The single task `CoordinatedInsertion` is then in charge of the close synchronization and information exchange between the two actuators, at lower levels. These two programmings are possible with our language and model.

Common pre- and postconditions in a sequence. The code included in primitive tasks contains *waitings* for the postconditions, that are immmediately followed by the exit of the task. On the other hand the preconditions of a task are also translated into *waitings*. It might sometimes occur that two tasks in a sequence have post- and preconditions in common, which can lead to redundant condition verification.

For example, in a contour following application such as the one presented below, a sequence:

```
SEQ( Approach, FollowContour )
```

includes two tasks:

- `Approach`, which has "contact reached" as postcondition,
- `FollowContour`, that has "contact reached" as precondition.

When used in sequence, these tasks will lead to a redundant verification of the fact that contact is reached.

On the other hand, these tasks can be useful as independent tasks, with full pre- and postcondition specification. For example, each one could be used independently from the other:

- the `Approach` task, can be used to gather sensory information through touching the environment, while groping one's way in case of otherwise insufficient sensing possibilities ;
- the `FollowContour` could be used not immediately after the `Approach` one: e.g. other tasks could be achieved before them in the sequence ; the meaning of checking the preconditions then is to make sure that we are still at contact, which might have changed in the meantime.

Therefore, each task should be defined as completely as possible. The possible redundancies in particular sequencings in applications would better be detected by the translator, where optimization should take place, e.g. by avoiding to produce redundant code.

4.5 A Contour Following Task

4.5.1 Script

This last example is concerned with a contour following problem, stated in [Simon e.a.91] and in [Samson e.a.90]. It is a complex hybrid position/force control task which consists in following the contour of an object according to a given contouring direction, a given force (normal to the plane of the contour, and of known intensity), and a given velocity. This task is carried out until an exception (such as a joint limit or an obstacle) is encountered, or until the contour has been explored all the way around.

This task is complex enough to provide a good support for specification. It requires many preconditions and postconditions, the actions to perform involve the use of many sensors, and it is necessary to generate events towards the outside world. We give in the following the main design steps of the related robot-tasks. Appendix C, taken from [Simon e.a.91], provides with all the related mathematical issues.

4.5.2 Related Object Instances

MODELS

Instanciation of an Object of the Class *Task-function*

Subclass TF

Attributes

robot = R_1

dimension = *6*

maxtime = *10 s*

$\lambda(t)$= *0*

initial condition = *measured*

Methods

symbolic computation: cancelled

coherence checking:cancelled

Continuation: redundant tasks

Subclass TF.RT

Attributes

working space =SE_3

e_1

h_s

W

$\alpha = 1$

Methods

Computation of $e = W^{\dagger}e_1 + \alpha(I - W^{\dagger}W)\frac{\partial h_s}{\partial x}$

Continuation: with sensors, of special type

Subclass TF.RT.WS.SPE ; object R1.Contour

Attributes

sensor = S_1

measurements: f ; $q, \dot{q}$ *for robot* R_1

imported parameters:

- M_0^6 *(imported from* MOD.KIN.R1.Contour*)*
- $J_{L_{|0}}$ *(imported from* MOD.TASK.JAC.R1.Contour*)*

other inputs

- *plane normal* N_p
- *desired normal force* f_d
- *desired velocity* v_d
- *initial contact point* $x_d(0)$

Methods

computation of e_1, g_s^r, e*:*

* *normal vector computation: equations (19), (18), (17) ;*
* *task frame computation: equations (20), (21) ;*
* *desired position tracking: equations (22), (23), (24) ;*
* *primary task: equation (11) ;*
* *secondary task: equations (28), (29), (30), (27) ;*
* *task function: equation (16).*

Instanciation of Objects of the Class *Models*

Subclass MOD.DYN.KE ; object R1.contour

Attributes

robot $=R_1$

model_choice=diag $(m_{11}, m_{22}...)$

measurements: q *for robot* R_1

Methods

computation code of $\hat{M}$*, with model_choice=diag*

Subclass MOD.DYN.ETC ; object R1.contour

Attributes

model_choice=0

Subclass MOD.KIN.DIR ; object R1.contour

Attributes

measurements: q for robot R_1

concerned_frame $= F_6$

expression_frame $= F_0$

Methods

computation of $M_0^6 = \begin{pmatrix} R_0^6 & T_0^6 \\ 000 & 1 \end{pmatrix}$

Subclass MOD.DM.DIR ; object R1.contour

Attributes

computing_choices= 1, 2, 3

Methods

1. *computation of* J_L *(translation)*
2. *computation of* J_R *(rotation)*
3. $J = \begin{pmatrix} J_L \\ J_R \end{pmatrix}$; *computation of* det *(J)*

Subclass MOD.DM.INV ; object R1.contour

Attributes

det *(J),imported from* MOD.KIN.DIR.DM.R1.contour

threshold σ

error_parameter=false

Methods

if |det *(J)*| $-\sigma > 0$

- *then, computation of* J^{-1} ;
- *else, error_parameter=true.*

Subclass MOD.TASK.TV ; object R1.contour

Attributes

R_0^w, *imported from* TF.RT.WS.SPE ; object .R1.Contour

J_L, *imported from* MOD.DM.DIR.R1.contour

measurements=$\dot{q}$

Methods

computation of $j_t = R_0^{wT} J_{L_{|0}} \dot{q}$

Remark

In this example, the objects MOD.TASK.JAC.DIR.R1.contour and MOD.TASK.JAC-.INV.R1.contour are chosen identical to objects MOD.DM.DIR.R1.contour and MOD.DM.-INV.R1.contour respectively.

Instanciation of an Object of the Class *Control*

Subclass CONT

Attributes

robot=R_1

task_function_selection=TF.RT.WS.SPE.R1.Contour

scalar positive gain values k *and* μ

matrix gains G *and* D

used models: MOD.DYN.KE.R1.contour *and* MOD.DYN.ETC.R1.contour

Continuation: Newton-type control

Subclass CONT.NEW ; object R1.contour

Attributes

B=*J_model:* MOD.TASK.JAC.DIR.R1.contour

A=J^{-1}*_model:* MOD.TASK.JAC.INV.R1.contour

C=j_t*_model:* MOD.TASK.TV.R1.contour

$F = 0$

Methods

$\Gamma = -k\hat{M}AG(\mu De + B\dot{q} + C) + \hat{N} - F$

Instanciation of Objects of the Class *Observers*

class OBS: *attribute robot*= R_1

Subclass OBS.TF ; object R1.contour.wdog

Attributes

task_function= TF.RT.WS.SPE.R1.contour

threshold=maxtime

monitored parameter= elapsed_time

Methods

running time generation

comparizon to maxtime

if greater, alarm generation

Subclass OBS.MOD.MON ; object R1.contour.sing

Attributes

model= MOD.TASK.JAC.R1.contour

threshold $= \epsilon$

monitored parameter= $|\det(\frac{\partial e}{\partial q})|$

Methods

comparizon to ϵ

if lower, alarm generation

Subclass OBS.VAR ; object R1.contour.joint_limit

Attributes

monitored variables=$q_1 ... q_6$

parameters min/max =$q_1^m, q_1^M ... q_6^m, q_6^M$

thresholds $\sigma_1 ... \sigma_6$

Methods

comparizons of distances to joint limits to thresholds

if lower, alarm generation

Subclass OBS.VAR ; object R1.contour.back_to_initial

Attributes

monitored parameters=$q_1 ... q_6$

initial contact point $x_d(0)$

thresholds σ_b

current x value (imported from MOD.KIN.DIR.R1.contour*)*

Methods

test $||x - x_d(0)|| - \sigma_b > 0$

if lower, alarm generation

Instanciation of an Object of the Class *Resources*

Subclass RE.SE.FI.RO ; object.R1.force

Attributes

calibration matrix M

Expression frame F_S / F_{06}

Measurements=$s_1 ... s_6$

Range of the measured forces f_m

Methods

Computation of f_m *in* F_s*:*

$F = M(s_1 ... s_6)^T$

expression de f_m *in the desired expression frame*

The *Robot-Task* object

Before giving the Robot Task skeleton, let us briefly come back to the associated task description. The required behavioral information are the following:

- before starting the control task execution, it is necessary to know the initial contact position, which needs to check the pre-condition CONTACT_DETECTION.
- upon reception of this pre-condition signal, the unique control task CONT.NEW-.R1.contour, is started.
- the execution of the control task must be monitored. Observation functions are thus executed in parallel. One associates a function to each critical signal involved in the control computation. Some information are reached through the activation of a physical sensor (exec JOINT_LIMIT_OBSERVATION()()). One software signal is to be observed to supervise the occurence of a singularity raised during the computation of the JACOBIAN.
- The task is over, when the arm is back to its initial position (post-condition). That is to say once the overall contour of the object has been followed.

The skeleton of this contour following task is given down below.

```
% waiting for precondition(s)
[
   % Force sensor for contact detection.
   trap SENSOR_1 in
   [
      exec SENSOR_CONTACT_DETECTION ()();
   ||
      waiting CONTACT_DETECTION;
      exit SENSOR_1;
   ]
   end; % trap
   % starting the control algorithm

   trap END_TASK, ERROR in
   [
      exec CONT.NEW.R1.contour()();
   ||
      % watch out the outside coming signals
      [
         exec SENSOR_JOINT_LIMIT_OBSERVATION()();
      ||
         waiting JOINT_LIMIT;
         exit ERROR
      ]
   ||
      [
         exec SENSOR_JACOBIAN()();
      ||
         every  JACOBIAN do
```

```
            exec JACOBIAN_PROCESSING()();
          end
      ]
   ||
      % Sensor to detect the initial position ie
      % the  associated post condition
      trap SENSOR_4 in
[
   exec SENSOR_INIT_POSITION()();
||
   waiting INIT_POSITION;
   exit SENSOR_4
]
      end;
      emit WELL_DONE;
      exit END_TASK;
   ]
   end;
]
.
```

Important Remarks:

1. *waiting* is a macro-primitive made of some of the ESTEREL language basic primitives. See section 3 and [Rutten90] for a detailed presentation of this primitive.

2. Observation Function for pre-condition : this code may be factorised in a module called CONTACT_DETECTION_OBSERVATION. Actually, it corresponds to the general form of an observation function. Therefore it is possible to use such an observation function for pre-conditions signals, for post-conditions signals, as well as for observation signals associated to a control law. Nevertheless, it is important to instanciate three different types of modules for observation functions, one for each possible processing: modify slightly the task, kill the task, or kill the application (cf section 2.2.6).

3. Here, the example requires only one pre-condition signal. When many signals are expected, as many modules as observation functions should be generated. All these modules are executed in parallel in order to make *all* the pre-conditions being satisfied before the execution of the control law.

4. the possible processings corresponding to observers are the following:

   - JACOBIAN induces a processing of type 1. Upon each reception of the signal JACOBIAN, an adequate processing is executed without interrupting the task.
   - JOINT_LIMIT implies a processing of type 2, that is to say the task must be killed.

Remark: It would be necessary to refine the error handling when a processing of type 2 is required. Indeed, the task abortion has to be managed properly and the error broadcasted in the whole application:

let us consider for example, the following sequence of robotics tasks "T_1 ; T_2 ; T_3". Each individuel task has its behaviour encoded in ESTEREL. Let T_1 behaves

uncorrectly. The error recovery scheme requires a processing of type 2. According to the ESTEREL features, task T_2 must start simultaneously to the end of T_1, and must in the same instant know exactly the current state of the robotics system. Therefore it is necessary to forecast the consequence of an error. A way to propagate this error must also be available. Otherwise the behaviour of the global system could not be predictable. In order to meet this requirement, the above skeleton must be slightly modified, and error recovery schemes will be studied in the near future. New exception handling mecanisms are to be extended to insure that processing of type 2 and of type 3 will be properly used.

If a task requires a processing of type 2, (the task is killed) two cases may be encountered: either the application can be carried on, because this task was not critical for the application and because the other tasks may be executed in order to overcome the error, or the error encountered leads to a processing of type 3. In any case, the error must be propagated to upper levels: this may imply little changes in the task skeleton.

5 Concluding Remarks

The work reported in this paper has to be understood as a preliminary study, a kind of prototype of the proposed approach. Several concepts still require to be validated, and the approach in itself is indeed open to various improvements. Let us mainly mention:

The object-oriented model At the present stage of evolution, it has been neither implemented, nor even simply expressed using any existing object-oriented language. In order to validate the modelling approach, this should be achieved in the framework of an actual experimental robotics system. On an other hand, the design of the required man-machine interface has to be coherent, not only with the object modelling, but also with the other programming levels needed in the Open Robot Controller, such as defined in [Simon e.a.91].

The robot-tasks design As stated previously, error processing and exception handling should be refined: as evoked in section 3.6, it is necessary to precisely define the various required error types and the related reaction levels. In particular, what is concerned with a *local* behaviour (processing internal to the robot-task) has to be distinguished from what has to be expressed in the *global* sequencing (for example assigning an alternative resource in case of failure). A tentative in this direction was given in section 3.6.

Another point is that robot-tasks instanciation and module-tasks design are not independent steps. It is important to precise how the design stage of robot-tasks is related to both the general object-oriented model and the definition of communication and temporal characteristics intra and inter module-tasks.

The application programming The implementation of a prototype of the translator was made using PROLOG II MALI [Le Huitouze88], that allowed a relatively straightforward implementation of the grammar. Further versions of it may be done using tools like **lexx** and **yacc**, in order to get a better integration within the Open Robot Controller (ORC) environment. A possible improvement is the introduction of parameters passing in the tasks, that would enhance the language expressivity. It has to be done in relation with the object model and the precise definition of its methods, and the external procedures called by **exec** statements. The possibility of optimizing the produced code should be studied, in terms of the automata produced in turn by ESTEREL, considering the number of states as well as the size of the *C* code implementing it. Finally, the control primitives of the language might be enriched in order to take other temporal constraints into account, e.g. specifying the duration of an execution.

Further perspectives consist in using this task-level interface in order to study the connection of an A.I. plan generator to the Open Robot Controller, or even its integration into the environment. The robot-tasks are here defined in terms of preconditions, observations and postconditions, which correspond to the abstraction level of the robots representation formalisms in A.I. planning. A planner could produce plans for an application, from an expression of the goals to be met ; this plan, with a form possibly different from what was presented here, should then find a translation into ESTEREL allowing its execution on the ORC.

From another point of view, it would be interesting to study the possibility of designing other kinds of specific user-oriented programming languages. They should keep the same characteristics (i.e. no need for knowing ESTEREL, the execution machine or the task-function and control theory), while extending to other areas, in particular the non-manufacturing robotics.

References

[André e.a.91] C. André, J.P Marmorat, J.P Paris, *Execution Machines for* ESTEREL, in Proceedings of European Control Conference (ECC 91), Grenoble, July 1991.

[Berry e.a.88] G. Berry, G. Gonthier: *The Synchronous Programming Language : Design, Semantics, Implementation*, INRIA Research Report No 842. 1988.

[Berry89] G. Berry, *Real-time programming: special purpose or general purpose languages*, in Proceedings of the IFIP World Computer Congress, San Francisco, 1989 (Invited talk).

[Borrelly e.a.90] J.J. Borrelly, D. Simon, *Propositions d'Architecture de Contrôleur Ouvert pour la Robotique*, INRIA Research Report No 1304, October 1990.

[Brandin e.a.91] B.A. Brandin, W.M. Wonham, B. Benhabib, *Discrete Event System Supervisory Control Applied to the Management of Manufacturing Workcells*, in ICAPE 91, Tennessee, USA, August 1991.

[Chedmail 90] P. Chedmail, *Synthèse de robots et de sites robotisés ; modélisation de robots souples*, Thése de doctorat d'état, ENSM, Nantes, July 1990.

[Chaumette90] F. Chaumette, *La relation vision-commande : théorie et application à des tâches robotiques*, Thése, Université de Rennes 1, July 1990.

[Coste-Manière89] E. Coste-Manière, *Utilisation d'Esterel dans un contexte asynchrone : une application robotique*, INRIA Research Report No 1139, Dec. 1989.

[Coste-Manière e.a.90] E. Coste-Manière, B. Faverjon, *A Programming and Simulation Tool for Robotics Workcells*, International Conference on Automation, Robotics, and Computer Vision, Singapore, Septembre 1990.

[Dombre e.a.88] E. Dombre, W. Khalil, *Modélisation et commande des robots*, Hermès Plubl., France, 1988.

[Espiau e.a90] B. Espiau, E. Coste-Manière, *A Synchronous Approach for Control Sequencing in Robotics Applications*, in Proc. IEEE International Worshop on Intelligent Motion, p 503-508, Istanbul, August 20-22 1990.

[EsterelV3] ESTEREL *V3 Compiler Overview*, in ESTEREL V3 Documentation, CISI Ingénierie, 1989.

[Gonthier88] G. Gonthier: *Sémantiques et modèles d'exécution des langages réactifs synchrones; application à* ESTEREL, Thesis, University of Paris-Orsay, 1988.

[Harel e.a.85] D. Harel, A. Pnueli: *On the development of Reactive Systems*, The Weizmann Institute of Science, Israel, January 1985.

[Faverjon86] B. Faverjon, *Object Level Programming of Industrial Robots*, IEEE International Conference on Robotics and Automation. April 7-10 1986.

[Faverjon89] B. Faverjon, *Hierarchical Object Models For Efficient Anti-Collision Algorithms*, 1989 IEEE International Conference on Robotics and Automation. May 14-19 1989.

[Gasmi90] B. Gasmi, *SAFIR, Système d'Assemblage Flexible Intelligent multi-Robots*, Thesis ENSAE, Toulouse November 1990.

[Le Huitouze88] S. Le Huitouze, *Mise en œuvre de* PROLOG II / MALI, Thèse de Doctorat d'Informatique de l'Université de Rennes I, 9 December 1988.

[de Mello e.a91] L.S. Homem de Mello, A.C. Sanderson, *A Correct and Complete Algorithm for the Generation of Mechanical Assembly Sequences*, IEEE trans. on Robotics and Automation, Vol 7, No 2, April 1991.

[Merlet87] J.P. Merlet, *C-surface theory applied to the design of an hybrid force-position controler*, IEEE International Conference on Robotics and Automation, Raleigh, 1987.

[Merlet91] J.P. Merlet, *Les Robots Parallèles*, Hermès Publ., France, 1990.

[Paris91] J.P. Paris, *Communications Synchrones Asynchrones. Application à* ESTEREL. Thesis to appear in June 1991.

[Rives e.a.89] P. Rives, B. Espiau, F. Chaumette: *Visual servoing based on a task function approach*, 1st International Symposium on Experimental Robotics, Montreal, Canada, June 1989.

[Roy e.a.89] V. Roy, R. de Simone: *An* AUTOGRAPH *primer*, INRIA Research Report No 111, October 1989.

[Rutten90] E.-P. Rutten, *Représentation en logique temporelle de plans d'actions dotés d'une structure de contrôle impérative; application à l'assistance à l'opérateur en téléopération*, Thèse de Doctorat d'Informatique, Université de Rennes I, July 1990.

[Rutten e.a.90] E. Rutten, J.C. Paoletti, G. André, L. Marcé, *A task-level language for operator assistance in teleoperation*, in Proceedings of the International Conference on Human Machine Interaction and Artificial Intelligence in Aeronautics and Space, Toulouse, September 1990.

[Samson87] C. Samson: *Une approche pour la synthèse et l'analyse de la commande des robots manipulateurs*, INRIA report No 669 (IRISA report No 356), May 1987.

[Samson e.a.90] C. Samson, B. Espiau: *Application of the task-function approach to sensor-based control of robot manipulators*, IFAC Tallinn, URSS, August 1990.

[Samson e.a.91] C. Samson, M. Le Borgne, B. Espiau: *Robot control: the task-function approach*, Oxford University Press, 1991.

[Simon e.a.91] D. Simon, A. Joubert, *ORCCAD: Towards an Open Robot Controller Computer Aided Design System*, INRIA Research Report No 1396, Feb. 1991.

[deSimone e.a.89] R. de Simone, D. Vergamini: *Aboard* AUTO, INRIA Research Report No 112, October 1989.

A The ESTEREL Language

A.1 Overview

Esterel ([EsterelV3,Gonthier88]) is thus the selected language for the intermediary programming level of applications. It is a high level, imperative language especially designed for writing reactive programs, i.e. communicating programs the behaviour of which depends on the ordering of inputs and outputs events. An important characteristic of reactive systems is their intrinsic *determinism*: a reactive system determines a sequence of output signals from a sequence of input signals in an unique way. Classical programming tools are not well-suited to reactive systems programming: automata-based systems lack high-level parallel programming primitives while asynchronous languages do not respect the intrinsic determinism of reactive systems.

The main ESTEREL feature is *synchrony*: the synchrony assumption states that a response to an input is instantaneous. This assumption implies that all execution times are equal to zero.

In ESTEREL parallel processes communicate without delay by signal broadcasting. The modular structure of ESTEREL allows also a hierarchical programmation. Two operators are essential: the "||" operator leads to an explicit parallelism. The branches of a 'parallel' statement start simultaneously (a 'parallel' terminates synchronously with the last termination of its branches) ; when using a ";" the programmers express a *sequence*, which means that the second statement of a sequence is performed exactly when the first statement terminates.

An ESTEREL program communicates with its environnement via *signals*. They are used both as inputs and outputs. They are assumed to be *broadcasted* instaneously among processes: signals are received by the different parts of the program within the same time (instant). Two kinds of information are broadcasted: *values* that are permanent, and *signal tops* that are intermittent. The former are available in expressions, the latest act as *control information* to be handled by ESTEREL control structures. The occurence of input signals defines the execution time of the program. Out of these occurences, the program has no activity. Time is handled in a rigorous way since the 'absolute' time doesn't exist in ESTEREL any signal being considered as an independent 'time unit'. Time manipulation primitives (for example a watchdog) can be uniformly used for all signals. This concept is the one of *multiform time*.

Local signals can be used for communication and synchronization purposes with other submodules. All signals are treated as messages, regardless of their hardware or software origin. Broadcasting simplifies process communication and improves modularity.

The synchrony assumption allows to establish a new form of process communication, the *instantaneous dialogue*.

As a programming language, ESTEREL is mathematically well-defined. Its implementation is linked to a rigourous semantics ([Berry e.a.88]).

The compiler transforms an ESTEREL program into a *finite determinist automaton*. Process scheduling and communication are performed at the compiling stage. The compiler transforms parallel programs into an equivalent sequential finite automaton. ESTEREL induces a good programming style (parallel language) and leads to execution efficiency (automaton). The translation of programs to automata has a major

advantage: it authorizes *automatic proofs* of the resulting automata to be performed: a property of the automaton that can be proved is associated to a precise property of the program. We use a verification system for parallel and communicating processes, the AUTO ([deSimone e.a.89]) system, to prove that the behaviour of the reactive system is the expected one. AUTO and AUTOGRAPH ([Roy e.a.89]), a graphical tool which draws the generated automaton, use experiment verification principles such as the quotient of an automaton by an observational criterion. Checked properties may therefore be infirmed or confirmed. After the allowed verifications, one can be sure of the logical correctness of the program.

A.2 The main primitives

We do not give here a precise definition of the ESTEREL primitives, but we introduce enough of the language to be able to understand the examples given in section 3.5 and in the Annexe B.

The basic programming unit is a **module** made of a *declaration part* and a *statement part.*

- The declaration part includes **type**, **constant**, **procedures**, **tasks** and **input**, **output** signals.
- The statement uses primitive statement based on a mathematically well defined semantics.

A.2.1 Some imperative statements

nothing	dummy statement
;	sequence statement
||	parallel statement
if < *cond* > **then** < *stat* > **else** < *stat* > **end**	condition statement
loop < *stat* > **end**	infinite loop
trap < *ident* > **in** < *stat* > **end**	trap definition
exit < *ident* >	to exit from a defined trap

where < *stat* > is a list of various statements, < *ident* > is an identifier, and < *cond* > is a condition.

A.2.2 Signal handling

Since ESTEREL communicates with its environnement via signals, it is important to describe the statements which handle signals. In the following S can be a signal of any type.

emit S allows to emit a signal. This emission can be performed inside the module where it is defined as a local signal using the **signal** < *ident* > **in** < *stat* > **end** primitive. It can be emitted toward the environnement, and is then declared as an **output** signal.

For signal reception it is possible to:

- test for the presence of a signal in a current event:
  present S **then** < *stat*1 > **else** < *stat*2 > **end**

- wait for a signal to occur: **await** S ;
- define a time limit for the execution of the statement, for example:

```
do
            < stat >
watching    S
```

- add a timeout close to a watchdog: the action specified after the timeout is executed if the time limit is reached before termination of the body:

```
do
            < stat >
watching    S
timeout     < stat >   end
```

- iterate the execution of actions whenever a signal is received:

```
loop                                    every   S          do
        < stat >            or                  < stat >
each S                                  end
```

- await many signals: this is the *multiple await*

```
await
        case < S1 > do
        case < S2 > do
        case < Si > do
end
```

ESTEREL includes a powerful exception handling mechanism that provides escapes from control block structures. This includes facilities for exception raising, recovering, and for exception handling. The exceptions are completely compatible with the ESTEREL parallelism, permitting process abortion, and either concurrent or preemptive treatment of simultaneous raised exceptions.

```
trap     < ident >   in
handle   < ident >   do
         < stat >
end
```

The action specified after the **handle** key word is executed if an exception is raised (with the **exit** key word).

Finally the new **exec** statement must be mentionned. This primitive deals with the execution of tasks which are not instantaneous i.e. *asynchronous tasks*. After their declaration, tasks are started using the blocking primitive:

```
exec TASK ;
```

Its mechanism is detailed in [Paris91]. This primitive is basic in the robotics context.

B Naive implementation of the first example

B.1 RobotA activities

```
module ROBOTA_MAINLY:
input
   OBJECT_VICINITY_RAB,
   OBJECT_IN_GRIP,
   LEFT_HAND_VICINITY,
   OBJECT_IS_IN_LEFT_HAND,
   RES1_FREE,
   RES2_FREE;
output
   RA_TRAJ_TOWARD_B,
   GRIPPING_ON_BELT,
   TRAJECTORY_TRACKING_TOWARD_LEFT_HAND,
   PUTTING_THE_OBJECT_IN_LEFT_HAND,
   TEST1,
   P1_SEM,
   V1_SEM,
   TEST2,
   P2_SEM,
   V2_SEM;

loop
   copymodule RA_FOLLOWING_A_TRAJ_TOWARD_B;
   copymodule REQUEST[ signal TEST1/TEST_RESOURCE,
      RES1_FREE/RESOURCE_FREE,
                              P1_SEM/P_SEM    ];
   copymodule GRIPPING_ON_BELT;
   emit V1_SEM;
   copymodule TRAJ_TRACKING_TOWARD_LEFT_HAND;
   copymodule REQUEST[ signal TEST2/TEST_RESOURCE,
      RES2_FREE/RESOURCE_FREE,
                              P2_SEM/P_SEM ];
   copymodule PUTTING_THE_OBJECT_IN_LEFT_HAND;
   emit V2_SEM;
end % of loop
.
```

B.2 RobotB activities

```
module ROBOTB_MAINLY:
input
   RB_NEAR_LH,
   ASSEMBLY_GRIPPED,
   DROPPING_IN_A_BASKET,
   RES3_FREE;
output
   RB_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND,
   GRIPPING_THE_ASSEMBLY,
   OBJECT_DROPPED,
   TEST3,
   P3_SEM,
```

```
   V3_SEM;

loop
   copymodule R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND;
   copymodule REQUEST[ signal TEST3/TEST_RESOURCE,
      RES3_FREE/RESOURCE_FREE,
      P3_SEM/P_SEM];

   copymodule GRIPPING_THE_ASSEMBLY;
   emit V3_SEM;
   copymodule DROPPING_OBJECT_IN_A_BASKET;
end % of loop
.
```

B.3 BeltA Control

```
module BELT_CONTROL:
input
   VICINITY_OF_B,
   P1_SEM,
   V1_SEM,
   TEST1;
output
   BELT_MOTION,
   RES1_FREE;

loop
   copymodule BELT_FORWARD;
   copymodule SEM[signal P1_SEM/ P_SEM,
 RES1_FREE/RESOURCE_FREE,
 TEST1/TEST_RESOURCE,
 V1_SEM/V_SEM];
end % of loop
.
```

B.4 LeftHand control

```
module LH_HANDLER:
input
   V2_SEM,
   P2_SEM,
   TEST2,
   V3_SEM,
   P3_SEM,
   TEST3;
output
   RES2_FREE,
   RES3_FREE;

loop
   copymodule SEM[signal P2_SEM/P_SEM,
 RES2_FREE/RESOURCE_FREE,
 TEST2/TEST_RESOURCE,
```

```
 V2_SEM/V_SEM];
   copymodule SEM[signal P3_SEM/P_SEM,
 RES3_FREE/RESOURCE_FREE,
 TEST3/TEST_RESOURCE,
 V3_SEM/V_SEM];

end % of loop
.
```

B.5 The synchronization modules

```
module SEM:
input
   V_SEM,
   P_SEM,
   TEST_RESOURCE;
output
   RESOURCE_FREE;
%loop
   trap RESOURCE in
      await immediate P_SEM do
 exit RESOURCE
      end
   ||
      loop
 emit RESOURCE_FREE
      each TEST_RESOURCE
   handle RESOURCE do
      await immediate V_SEM;
   end % of trap
%end % of loop
.

module REQUEST:
input
   RESOURCE_FREE;
output
   P_SEM,
   TEST_RESOURCE;

emit TEST_RESOURCE;
await immediate RESOURCE_FREE;
emit P_SEM;
.
```

B.6 The Application

```
module ACTION:

input
   OBJECT_VICINITY_R1B,
   OBJECT_IN_GRIP,
   LEFT_HAND_VICINITY,
   OBJECT_IS_IN_LEFT_HAND,
```

```
   R2_NEAR_LH,
   ASSEMBLY_GRIPPED,
   DROPPING_IN_A_BASKET,
   VICINITY_OF_B;

relation
   OBJECT_VICINITY_R1B #
   OBJECT_IN_GRIP #
   LEFT_HAND_VICINITY #
   OBJECT_IS_IN_LEFT_HAND #
   R2_NEAR_LH #
   ASSEMBLY_GRIPPED #
   DROPPING_IN_A_BASKET #
   VICINITY_OF_B;

output
   R1_TRAJ_TOWARD_B,
   GRIPPING_ON_BELT,
   TRAJECTORY_TRACKING_TOWARD_LEFT_HAND,
   PUTTING_THE_OBJECT_IN_LEFT_HAND,
   R2_FOLLOWING_A_TRAJ_TOWARD_LEFT_HAND,
   OBJECT_DROPPED,
   GRIPPING_THE_ASSEMBLY,
   BELT_MOTION;

signal
   RES1_FREE,
   RES2_FREE,
   RES3_FREE,
   TEST1,
   P1_SEM,
   V1_SEM,
   TEST2,
   P2_SEM,
   V2_SEM,
   TEST3,
   P3_SEM,
   V3_SEM
in

   loop
      copymodule ROBOTA_MAINLY;
   ||
      copymodule BELT_CONTROL;
   ||
      copymodule ROBOTB_MAINLY;
   ||
      copymodule LH_HANDLER;
   end % of loop
end % of signal
.
```

C The Main Classes

Preliminary Remark:

We propose in this appendix a tentative set of object-oriented models corresponding to the algorithmical and physical entities as defined in section 2.2. Their presentation follows the successive levels of every class hierarchy given in figures 3 to 6. All related mathematical details may be found in ([Samson e.a.91]). As stated in the report, this first structuration is done under a naive form and should be reconsidered when using an adequate object-oriented language.

Class *Task-function*

An object instance in this class takes measurements as inputs and provides with the value of the vector e.

Subclass TF

Attributes

$e(q,t)$
concerned robot
needed measurements
dimension n
maxtime T
$\lambda(t)$
$y(t)$ *(default:* $q(t)$ *filtered)*
initial condition q_0
other parameters
selection of optional methods

Methods

$e(q,t) = e(q,t) + \lambda(t)(q - y(t))$
option: symbolic computation of $e-$*derivatives*
option: coherence checking $e(q_0) = 0$*?*

Continuations: redundant (RT); gradient (GRA);
trajectory tracking (TT); others (OT)

Subclass TF.RT

Attributes

working space x
subdimension m
e_1
h_s *(secondary cost function)*
W *(type:* $m \times n$ *full-rank matrix)*
α *(type: positive scalar)*
$\frac{\partial h_s}{\partial x}$*: given or computed*

Methods

option: symbolic computation of $\frac{\partial h_s}{\partial x}$

Computation of $e = W^{\dagger}e_1 + \alpha(I - W^{\dagger}W)\frac{\partial h_s}{\partial x}$

Continuation: with sensors (WS); without sensors (NS)

Subclass TF.GRA

Attributes

$h(q,t)$ *to be minimized*

$\frac{\partial h}{\partial q}$*: given or computed*

Methods

option: symbolic computation of $\frac{\partial h}{\partial q}$

$e(q,t) = \frac{\partial h}{\partial q}$

Subclass TF.TT

Continuations: in SE_3 (SE3); in joint space (JS)

Subclass TF.OT

Attributes

task-function parameters

Methods

computation algorithm for $e(q,t)$

Subclass TF.RT.WS

Attributes

required sensor measurements, s

p *(dimension of* s*)*

D *(type* $m \times p$ *matrix)*

reference value σ_r *(type* $m-$*dimensional vector)*

name of the trajectory file t_d *to be tracked*

Methods

working space $\{x\}=\{\bar{r}\}$

$e_1 = Ds - \sigma_r$

computation of $\frac{\partial h_s}{\partial \bar{r}}(t_d)$

Continuations: proximity (PRO); force (FO); vision (VI); special (SPE)

Subclass TF.RT.NS

Attributes

type of main task:in SE_3*; other*

in SE_3*: concerned_frame*

in SE_3: tracked positions and used kinematics

in SE_3: attitude error parametrization

in SE_3: name of the trajectory file t_d to be tracked

Methods

computation of e_1:

in SE_3: computation of $\begin{pmatrix} x(q) - x_r(t) \\ E_a(q,t) \end{pmatrix}$

computation of W

Continuations: avoidance of: joint limits (JL); obstacles (OBS); singularities (SING); others (OT)

Subclass TF.TT.SE3

Attributes

concerned_frame

tracked positions and used kinematics

attitude error parametrization

name of the trajectory file t_d to be tracked

Methods

computation of $\begin{pmatrix} x(q) - x_r(t) \\ E_a(q,t) \end{pmatrix}$

Subclass TF.TT.JS

Attributes

name of the trajectory q_d to be tracked

Methods

computation of $e(q,t) = q - q_d(t)$

Subclass TF.RT.WS.FO

Attributes

concerned sensor

related frame

related parameters

calibration parameters

pre-processing parameters

α

desired force σ_r

Methods

computation of D

computation of W

Subclass TF.RT.WS.PRO

Attributes

concerned sensors
related frame
related parameters
calibration parameters
pre-processing parameters
α
desired virtual linkage

Methods

computation of s
computation of D
computation of W

Subclass TF.RT.WS.FO

Attributes

concerned sensor
related frame
related parameters
calibration parameters
pre-processing parameters
α
desired force σ_r

Methods

computation of D
computation of W

Subclass TF.RT.WS.SPE

Attributes

concerned sensor
related parameters

Methods

related specific computations

Subclass TF.RT.WS.VI

Attributes (see ([Chaumette90])

concerned sensor
target type
virtual linkage to be realized
used matrix D: $W\hat{L}$ *or* $W\hat{L}^{T+}$

α

Methods

computation of s

computation of W

Subclass TF.RT.NS.JL

Attributes

considered joint limits, $\{ q_i^{min}, q_i^{max} \}$, $i \in I$

Methods

computation of $h_s = \sum \frac{(q_i - \frac{1}{2}(q_i^{min} + q_i^{max}))^2}{(q_i^{max} - q_i^{min})^2}$

Subclass TF.RT.NS.SING

Attributes

considered singularities

method selection (manipulability, SVD,etc...)

Methods

selected computation method of h_s *(ex :*$\sqrt{det(JJ^T)}$*)*

Subclass TF.RT.NS.OBS

Attributes

method selection

required measurements

Methods

selected computation method of h_s

(maximization of a distance to obstacles)

Subclass TF.RT.NS.OT

Attributes

type of used cost function

required measurements

Methods

computation method of h_s

Remark

We did not detailed the mathematical expressions of most of the presented subclasses since the set of all possible cases is huge. All required information may be found in the indicated references.

Class *Control*

An object instance in this class provides with the current computed value of an array of control torques.

Subclass CONT

Attributes

robot=R_1

task_function_selection (object name in class TF.)

scalar positive gain values k *and* μ

matrix gains G *and* D

used models $\hat{M}$ *(object name in class MOD.DYN.KIN) and* $\hat{N}$ *(object name in class MOD.DYN.ETC)*

Methods

$\Gamma = -k\hat{M}AG(\mu De + B\dot{q} + C) + \hat{N} - F$

Continuations: Newton-type control (NEW); gradient-type control (GRA)

Subclass CONT.GRA

Attributes

$A = I_n$

$B = I_n$

$C = 0$

$F = 0$

Subclass CONT.NEW

Attributes

used model for $\frac{\widehat{\partial e}}{\partial q}$ *(object name in class MOD.TASK.JAC.DIR)*

used model for $\left(\frac{\widehat{\partial e}}{\partial q}\right)^{-1}$ *(object name in class MOD.TASK.JAC.INV)*

used models in $\hat{f}$ *(object names in classes MOD.TASK.TA and MOD.TASK.OT)*

used model for $\frac{\widehat{\partial e}}{\partial t}$ *(object name in class MOD.TASK.TV)*

Methods

assignation of A, B, C *to selected models*

computation of $F = \hat{M}\left(\frac{\widehat{\partial e}}{\partial q}\right)^{-1}\hat{f}$

Class *Models*

Subclass MOD

Attributes

concerned robot

Continuations: dynamics (DYN); kinematics KIN); differential motion (DM); task (TASK)

Subclass MOD.DYN

Continuations: kinetics energy matrix (KE); other terms (ETC)

Subclass MOD.KIN

Continuations: direct model (DIR); inverse model (INV)

Subclass MOD.DM

Continuations: direct model (DIR); inverse model (INV)

Subclass MOD.TASK

Attributes

task_function_selection (object name in class TF.)

Methods

computation of $\hat{f} = \widehat{\frac{\partial^2 e}{\partial t^2}} + \widehat{\dot{J}_T \dot{q}}$

Continuations: task-jacobian (JAC); task-velocity (TV); task acceleration (TA); other terms (OT)

Subclass MOD.DYN.KE

Attributes

model selection: $M(q)$, M *=constant,* $M(q$*,payload),* $M(q$*,identified* $\theta)$*...*

Methods

computation of the selected model

Continuation:

Subclass MOD.DYN.ETC

Attributes

model selections for gravity, Coriolis/centrifugal, friction forces

Methods

computation of the selected model $N(\hat{q}, \dot{q}, t$

Subclass MOD.KIN.DIR

Attributes

concerned_frame F_c

expression_frame F_e

measurement inputs q

outputs $R_e^c(q)$ *and* $T_e^c(q)$

Methods

computation of $M_e^c(q) = \begin{pmatrix} R_e^c(q) & T_e^c(q) \\ 000 & 1 \end{pmatrix}$

Subclass MOD.KIN.INV

Attributes

concerned_frame F_c

expression_frame F_e

selection of the criterion of solution choice:

removal from bounds, from singularities, proximity of the previous solution...

desired R_e^c *and* T_e^c

output q

Methods

computation of q *such that* $M_e^c(q) = \begin{pmatrix} R_e^c & T_e^c \\ 000 & 1 \end{pmatrix}$

Subclass MOD.DM.DIR

Attributes

concerned_frame F_c

expression_frame F_e

measurement inputs $q, \dot{q}$

computing selections

outputs: jacobian matrices, V, ω

Methods

according to selections, computation of:

$J_L(q)$ *(translation jacobian);* $J_R(q)$ *(rotation jacobian); concatenation* J*; det* (J)

$V(q, \dot{q})$ *(translation velocity);* $\omega(q, \dot{q})$ *(rotation velocity); concatenation*

Subclass MOD.DM.INV

Attributes

used direct differential model (object name in class MOD.DM.DIR)

threshold σ

det (J)

error_parameter=false

Methods

if $-det\ (J)-$ - $\sigma > 0$

- *then computation of* J^-1
- *else error_parameter=true*

Continuation:

Subclass MOD.TASK.TV

Attributes

model computation selection (symbolic, digital, estimated, zero, datafile...)

inputs q and t

output $\widehat{\frac{\partial e}{\partial t}}(q,t)$

Methods

computation of $\widehat{\frac{\partial e}{\partial t}}(q,t)$

Subclass MOD.TASK.TA

Attributes

model computation selection (symbolic, digital,estimated, zero, datafile...)

inputs q and t

output $\widehat{\frac{\partial^2 e}{\partial t^2}}(q,t)$

Methods

computation of $\widehat{\frac{\partial^2 e}{\partial t^2}}(q,t)$

Subclass MOD.TASK.OT

Attributes

model computation selection (symbolic, digital, zero)

inputs q and t

output $\widehat{\dot{J}_T\dot{q}}$

Methods

computation of $\widehat{\dot{J}_T\dot{q}}$

Subclass MOD.TASK.JAC

Attributes

model selections: robot jacobian (object names in MOD.DM.) or special task-jacobian

inputs q and t

output $\hat{J}_T$

Continuations (if special): direct (DIR); inverse (INV)

Subclass MOD.TASK.JAC.DIR

Attributes

det $(\hat{J}_T)$ *(optional)*

Methods

computation of $\hat{J}_T$; det $(\hat{J}_T)$ *(optional)*

Subclass MOD.TASK.JAC.INV

Attributes

threshold

model selection

output $\widehat{J_T^{-1}}(q,t)$

Methods

computation of selected model $\widehat{J_T^{-1}}(q,t)$ *(for example* $J_T^T(q,t)$*)*

test of det $(\hat{J}_T)$

Class *Robot-task*

We present here the robot-tasks briefly, since a detailed description of this class is given in section 2.2. Let us mention that a primitive robot-task RBT is only characterized by the simplicity of its own behaviour.

Subclass RBT

Attributes

pre-condition signals

post-condition signals

name of the control to be executed (object in class CONT.)

associated parameters

Methods

ESTEREL-*coded elementary behaviour (type 0) corresponding to:*

- *pre-conditions handling*
- *post-conditions handling*

activation of the asynchronous control task

Continuation: general robot-task (GEN)

Subclass RBT.GEN

Attributes

activated observers (object names in class OBS.)

associated parameters

Methods

ESTEREL-*coded behaviour corresponding to:*

- *complex pre- and post-conditions handling*
- *type 1 observers processing*
- *type 2 observers processing*
- *type 3 observers processing*

Class *Observers*

Subclass OBS.TF

Attributes

concerned task function (object name in class TF)

selected monitored parameters: $||e||$*; elapsed time*
related thresholds: emax; maxtime
names of associated events

Methods

running time counting
comparizon to maxtime
if greater, alarm generation
evaluation of $||e||-$ *emax*
if greater, alarm generation

Subclass OBS.SEN

Attributes

concerned sensor (object name in class RE.SE.)
sensor variables to be monitored
associated parameters and thresholds
names of associated events
other outputs

Methods

related computations
related tests
alarms and events generation

Subclass OBS.EXT

Attributes

concerned physical resource
physical signals to be monitored
associated thresholds and parameters
associated events and outputs

Methods

associated computations
alarm generation (example: motor failure detected by current monitoring)

Subclass OBS.VAR

Attributes

concerned control (object name in class CONT.)
list of monitored variables in the control expression (examples: q_3*;* $k(q, \dot{q}, t)$*)*
associated parameters
associated thresholds
associated events

Methods

associated computations

alarm generation

Subclass OBS.MOD

Attributes

concerned model (object name in class MOD.)

Continuations: monitoring (MON); estimation (EST)

Subclass OBS.MOD.MON

Attributes

monitored parameters(examples: $|\det(\frac{\partial e}{\partial q})|$; $\sigma_{min}(\frac{\partial e}{\partial q})$; $\frac{\sigma_{min}}{\sigma_{max}}(\frac{\partial e}{\partial q})$; $|\det(J)|$ *(robot jacobian))*

positivity; singularities

associated parameters

associated thresholds

associated events

Methods

associated computations

alarm generation

Subclass OBS.MOD.EST

Attributes

variables to be estimated (examples: $\frac{\partial e}{\partial i}$*; payload; inertial or kinematical parameters)*

associated tuning parameters

associated thresholds

associated events

Methods

estimation algorithms

alarm generation

Class *Trajectory Generation*

Subclass GT

Attributes

concerned robot

Continuations: in joint space (JS); in a sensory space (SEN); in SE_3 (SE3)

Subclass GT.JS

Attributes

duration T
initial value q_0^d
final value q_T^d
number of points n_p

Continuations: polynomial trajectory (POL); reference model (REF)

Subclass GT.SEN

Attributes

concerned sensor
duration T
initial value s_0^d
final value s_T^d
number of points n_p
output $s^d(i)$, $i = 1..n_p$

Methods

computation of $s^d(i)$, $i = 1..n_p$

Subclass GT.SE3

Attributes

duration T
position: initial and final values T_0^d, T_T^d
attitude: initial and final rotation matrices R_0^d, R_T^d
number of points n_p
output $T^d(i)$, $R^d(i)$, $i = 1..n_p$

Continuations: polynomial trajectory (POL); reference model (REF)

Subclass GT.JS.POL

Attributes

imposed parameters (velocity, acceleration, intermediary points...)
required accuracy

Methods

polynomial computations

Subclass GT.JS.RM

Attributes

type of reference model (2nd order, minimal time, mixed...)
associated parameters

Methods

reference model generation

Subclass GT.SE3.POL

Attributes

imposed parameters (velocity, acceleration, intermediary points...)

required accuracy

Methods

computation of rotation axis and angle

polynomial computations

Subclass GT.SE3.RM

Attributes

type of reference model (2nd order, minimal time, mixed...)

associated parameters

Methods

computation of rotation axis and angle

reference model generation

Class *Physical Resources*

This class is aimed to describe the working environment. We consider here the case of an assembly cell. Several examples of object-oriented modelling of such a workcell are encountered in the literature, mainly in the area of Computer-Integrated Manufacturing. This is why we give here only some guidelines for building such a model, recalling that the proposed structure has to be designed in strong connection with the previously described entities (cf fig 2).

Subclass RE

Continuations: machine-tools (MT); feeders (FE); conveyor belts (CB); assembly supports (ASS); robot manipulators (ROB); robot removable tools (ROT); sensors (SE); storages (STO)

Subclass RE.MT

Attributes

provided service = processing

type of tooling

associated parameters

associated measurements

location (grasping and ungrasping frames)

state

Methods

evaluation of the state: free/busy; progress of processing...

Subclass RE.FE

Attributes

provided service = feeding

capacity: type of provided parts

location (grasping frame)

state

Methods

evaluation of the state: part ready; empty...

Subclass RE.CB

Attributes

provided service = feeding; transport; motion

capacity: type of accepted parts

associated parameters

associated measurements

location (grasping and ungrasping areas)

state

Methods

evaluation of the state: free/busy; motionless/running; objects in transfer...

Subclass RE.ASS

Attributes

provided service = motion; reception; assembly

location (grasping and ungrasping areas)

state

Methods

evaluation of the state: free/busy; clamping/not

Subclass RE.ROB

Attributes

provided service = motion; transport; assembly

static description: Denavit-Hartenberg parameters; frames; inertia and mass parameters; reduction rates; joint bounds; actuators (current, torque and velocity limitations)

payload description: maximal allowed mass; type of the end effector (jaws, opening, tightening force)

performances: accuracy, transfer velocity, repeatability

location: basic frame; reachable space

state

monitoring events

Methods

model computations in relation with the class MOD
other computations: reachable space, aspects
opening/closing of the gripper
state evaluation ($(q; \dot{q})$; free/busy...)

Continuation: multi-purpose manipulators (MPM)

Subclass RE.ROT

Attributes

provided service = processing; transport; grasping.
(examples: screw-driver, drill, grinder, dedicated gripper...)
type of mechanical connection
associated parameters
associated measurements
location in the tool storage unit
state

Methods

evaluation of the state: current connection and location, availability
grasping method

Continuation: multi-purpose manipulators (MPM)

Subclass RE.SE

Attributes

provided service = information
type of provided information
associated parameters
outputs
state

Methods

local signal processing
calibration
evaluation of the state: free/busy; measurement available...

Continuation: removable (REM); fixed (FI)

Subclass RE.STO

Attributes

provided service = storage
associated parameters (type of accepted parts)
location (grasping and ungrasping areas)

Continuations: end (END); buffer (BUF)

Subclass RE.STO.END

Attributes

location (ungrasping area)

state

Methods

evaluation of the state: filling ratio; next free place...

Subclass RE.STO.BUF

Attributes

capacity

location (grasping and ungrasping areas)

state

Methods

evaluation of the state: stack state; component identification

Subclass RE.SE.REM

Attributes

type of mechanical connection

location in the sensor storage

Methods

grasping method

state updating: current connection and location

Continuation: multi-purpose manipulators (MPM)

Subclass RE.SE.FIX

Attributes

location: on a robot; in the cell

associated parameters

Subclass RE.(ROB, ROT or SE.REM). MPM

This seems to be a case where multiple inheritance is required. The related methods have to be carefully designed since their redefinition is allowed: for example "gripper opening/closing" should be transformed in "tool gripping/ungrasping". It is useless to provide with more details in this first approach.

Class *Parts*

A part is a physical object associated at any time with a physical resource.

Attributes

type: workpiece; tool (object name in class RE.ROT); sensor (object name in class RE.SE.REM). For workpiece:

invariant grasping parameters: acceptable jaws; allowed grasping frame; tightening force limits; overall size; mass; inertia...

maintained state:

- current linkage: type and associated parameters in a given frame (default= part_frame)

- current resource supporting the linkage

- part_frame location (/ cell or resource)

- current location uncertainty

- history of performed measurements

- history of performed transformations

Methods

automatic determination of current grasping context (preferred frame, needed accuracy...)

current state evaluation

Class *Cell*

Attributes

reference frame

list of the concerned resource instances

associated frames

static accessibility graph in the cell workspace

state of the cell: list of busy/free/out of order resources...

Methods

static: computation of optimal resources placement (cf ([Chedmail 90]))

static: computation of the accessibility graph

state evaluation

D A Contour Following Example

D.1 Notations

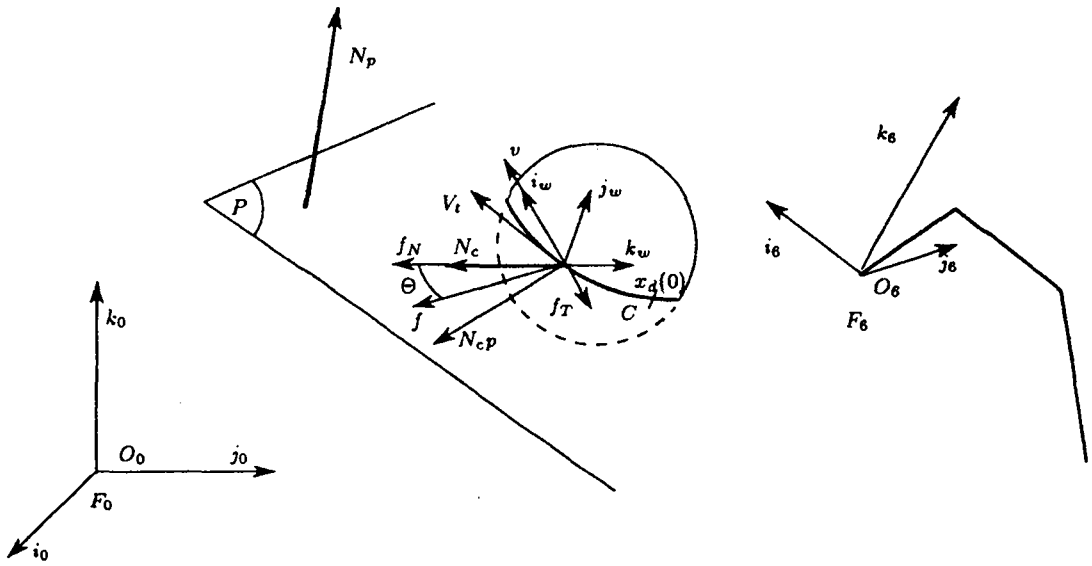


Figure 12: A contouring task

$F_0 = [O_0, i_0, j_0, k_0]$ fixed frame in the cartesian space
$F_6 = [O_6, i_6, j_6, k_6]$ frame fixed to the end effector, O_6 is located at the end-point of the tool
$F_w = [O_6, i_w, j_w, -N_c]$ task frame, defined with respect to the normal at the contour and the velocity of O_6
$M_0^6 = \begin{pmatrix} R_0^6 & T_0^6 \\ 000 & 1 \end{pmatrix}$ homogeneous transformation matrix from F_0 to F_6
R_6^w rotation matrix from F_6 to F_w
(with the subscript convention $v_{|i} = R_i^j v_{|j}$, $v_{|i}$ being the expression of the vector v in the frame F_i)

N_p	normal to the plane P	3*1
C_k	flag for contouring direction	integer
f_d	desired normal force	scalar
v_d	desired contouring velocity	scalar
e	error vector	6*1
$F = [f, \tau]$	contact screw	6*1
f	measured contact force	3*1
W	partition matrix	6*3
N_c	normal to the contour	3*1
α	secondary goal gain	scalar
$x_d(0)$	initial contact point	3*1
$x(t)$	position of O_6	3*1
$q(t)$	joint coordinates	6*1
d	distance between O_0 and the plane P	scalar
f_N	normal contact force	3*1
f_T	tangential contact force	3*1
$v_{\|0}$	velocity of O_6	3*1
V_t	desired velocity of O_6	3*1
N_{cp}	normalized projection of N_c on P	3*1
c_i	constraints defining the secondary goal	scalars
Γ	control torque	6*1
$\hat{M}$	estimated inertia matrix of the robot	6*6
J_q	estimated task Jacobian	6*6
j_t	estimation of $\frac{\partial e}{\partial t}$	6*1
$J_{L_{\|0}}$	translational Jacobian	3*6
$\hat{l}$	centrifugal and Coriolis forces	6*1
$\mu, k,$	scalar gains	
G, D	matrices of gains	6*6

D.2 From specification to control law

- 3 dimensioned error function to be regulated (error vector of the primary goal):

$$e_1 = \begin{bmatrix} < i_w, O_d O <_6> \\ < j_w, O_d O_6 > \\ < N_c, f > - f_d \end{bmatrix} \tag{11}$$

$< ., . >$ denoting the scalar product of vectors. O_d is the on-line computed point of the contour to be tracked with the velocity v_d
This formulation states that we wish to track the projection of O_d in the plane $[i_w, j_w]$ while regulating the measured force around f_d in the orthogonal direction given by N_c.
The sensed object is motionless with respect to F_0, so the derivative of the primary task-function with respect to time may be written as:

$$\dot{e_1} = \frac{\partial e_1}{\partial \bar{r}} \frac{d\bar{r}}{dt} = V_{6/0} \tag{12}$$

where $V_{6/0}$ is the (6×1) velocity screw of F_6 with respect to F_0. L is the (6×3) interaction matrix which states the variation of e_1 with respect to the relative

motion between the gripper and the target. $V6/0$ and L are computed at the contact point O_6. Assuming that the contour to follow is smooth enough to consider that the task frame is slowly varying, the interaction matrix associated with e_1 may be estimated by:

$$\hat{L} = \begin{bmatrix} i_w & j_w & -\lambda \hat{N}_c \\ 0 & 0 & 0 \end{bmatrix} \tag{13}$$

where λ is the unknown stiffness along the direction of N_c. A possible choice for a matrix function W such as $Range(W^T) = Range(L)$ and $L^T W^T > 0$ is therefore:

$$W = \begin{bmatrix} i_w^T & 0 \\ j_w^T & 0 \\ -\hat{N}_c^T & 0 \end{bmatrix} \tag{14}$$

and the partition matrices between the primary task-function and the secondary goal can be computed as (W^+ denotes the pseudo inverse of W):

$$W_1 = W^+ = \begin{bmatrix} R_0^w \\ 0 \end{bmatrix} \qquad W_2 = I_6 - W^+ W = \begin{bmatrix} 0 & 0 \\ 0 & I_3 \end{bmatrix} \tag{15}$$

- Computation of the global task function (expressed in F_0):

$$e = \begin{bmatrix} R_0^w \\ 0 \end{bmatrix} \begin{bmatrix} < i_{w|0}, T_0^6 - x_d > \\ < j_{w|0}, T_0^6 - x_d > \\ < k_{w|0}, R_0^6 f_{|6} > - f_d \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & I_3 \end{bmatrix} g_s^{\bar{r}} \tag{16}$$

where $g_s^{\bar{r}} = \frac{\partial h}{\partial \bar{r}}$ is the gradient of a cost function $h_s(\bar{r}, t)$ to be minimized and $\bar{r}$ is the location of the gripper in the cartesian space.

- Preliminary calculations
  -storage of the initial contact position $[O_0 O_d]_{|0}(0) = x_d(0) = G(q(0))$, G is the geometric model of the manipulator
  -computation of the distance $d = (O_0, P) =< N_{p|0}, x_d(0) >$

- Normal vector estimation:

$$\hat{N}_{c|0} = \frac{(R_0^6 f_{|6} - f_{T|0})}{||R_0^6 f_{|6} - f_{T|0}||} \tag{17}$$

The tangential force f_T can be computed by a "normal expert" assuming that a realistic model of friction is available [Merlet87] ; an other way to compute f_T might be:

$$f_{T|0} =< f_{|0}, v_{|0} > \frac{v_{|0}}{||v_{|0}||} \tag{18}$$

$v_{|0}$ being the measure of the velocity of the end point of the robot computed in F_0 given by

$$v_{|0} = J_{L|0}(q)\dot{q} \tag{19}$$

, $J_{L|0}(q)$ being the translational Jacobian of the manipulator, expressed in the frame F_0.

- Computation of the task frame:

$$i_w = \frac{v}{||v||} \qquad j_w = -\hat{N}_c \times i_w \tag{20}$$

where $\times$ denotes the cross product of vectors

$$R_0^w = \left[\begin{array}{ccc} i_{w|0} & j_{w|0} & -\hat{N}_{c|0} \end{array} \right] \tag{21}$$

- the tool follows the contour at a velocity v_d:
-projection of the normal N_c on the plane P (normalized):

$$\hat{N}_{cp|0} = \frac{\hat{N}_{c|0} - < N_{p|0}, \hat{N}_{c|0} > N_{p|0}}{||\hat{N}_{c|0} - < N_{p|0}, \hat{N}_{c|0} > N_{p|0}||} \tag{22}$$

-desired velocity of displacement:

$$V_{t|0} = (\hat{N}_{cp|0} \times N_{p|0}) v_d C_k \tag{23}$$

with $C_k = 1$ for clockwise contouring and -1 otherwise

- Position to track, in continuous time:

$$x_d(t) = x_d(0) + \int_0^t V_t(s) ds \tag{24}$$

V_t is computed in F_0. A stable implementation in discrete time is given by the following expression, with $(.)^k = (.)(k\Delta)$, Δ being the sampling period for the desired position computation loop:

$$x_d^{k+1} = (1-\beta) x_d^k + \beta x^k + \Delta V_t^k; 0 < \beta < 1 \tag{25}$$

- Cost function to minimize:

$$h_s = \frac{1}{2} \sum_i \alpha_i c_i^2, \alpha_i > 0 \tag{26}$$

$$g_s^r = \sum_i \alpha_i c_i \frac{\partial c_i}{\partial \bar{r}} \tag{27}$$

These functions are defined by the constraints c_i weighted by the scalars α_i.

- Constraints defining the secondary goal:
the k_6 axis of the tool remains parallel to the normal at the contact point:

$$c_1 = < i_6, \hat{N}_{c|0} > = 0 \qquad \frac{\partial c_1}{\partial \bar{r}} = \left[\begin{array}{c} 0 \\ i_6 \times \hat{N}_{c|0} \end{array} \right] \tag{28}$$

$$c_2 = < j_6, \hat{N}_{c|0} > = 0 \qquad \frac{\partial c_2}{\partial \bar{r}} = \left[\begin{array}{c} 0 \\ j_6 \times \hat{N}_{c|0} \end{array} \right] \tag{29}$$

the i_6 axis of the wrist keeps parallel to the plane P:

$$c_3 = < i_6, N_{p|0} > = 0 \qquad \frac{\partial c_3}{\partial \bar{r}} = \left[\begin{array}{c} 0 \\ i_6 \times N_{p|0} \end{array} \right] \tag{30}$$

- A general form of the control torque is:

$$\Gamma = -k\hat{M}J_q^{-1}G(\mu D\hat{e} + J_q\hat{\dot{q}} + j_t) + \hat{l} \qquad (31)$$

with $\hat{M}$ being an estimation of the inertia matrix of the manipulator, μ,G, k and D being gains, J_q an estimation of the Jacobian matrix $\frac{\partial e}{\partial q}$of the task, j_t being an approximation of the derivative of the task-function with respect to time $\frac{\partial e}{\partial t}$ and $\hat{l}$ includes an estimation of the Coriolis and centrifugal forces.
Hybrid position/force control tasks are generally performed at a rather low speed, with few dynamical effects, while force control requires a high sampling rate. Moreover, we assume that the estimation of the task frame is accurate enough to insure that $\frac{\partial e}{\partial q}J_0^{-1} > 0$. The following choices are thus made:
$\hat{M} = diag(M)$ with a slow sampling rate
$\hat{l} = 0$
$j_t = \frac{\partial e_1}{\partial t} = R_0^{wT}J_{L_{|0}}\dot{q}$ (giving feedforward only for the primary goal)
$J_q = J_{|0}$ (the basic Jacobian of the manipulator)
$J_q^{-1} = J_{|0}^{-1}$
These choices and the value of the gains will have to be verified by simulation.

- Events to be generated:
BACK_TO_INITIAL when a desired position is reached, SINGULARITY when the norm of the Task jacobian reaches a given threshold, JOINT_LIMIT when a specified joint limit is reached.

D.3 Module-Tasks to be used

Task name	Function	Input ports	Output ports
J0	basic Jacobian	q_i^k , $i = 1,6$	$J_{\|0}$ (6*6), $\|\|J_{\|0}\|\|$
Jinv0	basic inverse Jacobian	q_i^k , $i = 1,6$	$J_{\|0}^{-1}$ (6*6)
GM	Direct Geometric Model	q_i^k , $i = 1,6$	M_0^6 (4*4), $\bar{r}$ (6*1)
FN	Normal Expert	$f_{\|6}, v_{\|0}, R_0^6$	$f_{N_{\|0}}$ (3*1) ,R_0^w (3*3),$\dot{N}_{c_{\|0}}$ (3*1)
Calib	Calibration matrix of the force sensor	s_i , $i = 1,6$	$F_{\|6}$ (6*1)
Control	Control Torque vector	$e, \dot{q}, diag(M), J_{\|0}, J_{\|0}^{-1}$	Γ (6*1)
DiagM	diagonal elements of M	q_i^k , $i = 1,6$	$M_{ii}, i = 1,6$
TrackXd	Trajectory generator	$\dot{N}_{c_{\|0}}$	x_d^{k+1}
TF	task function	$M_0^6, R_0^w, x_d, f_{\|6}$	e (6*1), $\|\|e\|\|$
AAF	Anti-aliasing filter	$q_i, \dot{q}_i$, $i = 1,6$	$q_i^k, \dot{q}_i^k$, $i = 1,6$
OBS	observer	$x, \|\|J_0\|\|, \|\|e\|\|$	SING., JOINT., BACK.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399