Visual Conveyor tracking in High-speed Robotics Tasks

Theodor Borangiu

1. Introduction

The chapter presents two methods and related motion control algorithms for robots which are required to pick "on-the-fly" objects randomly moving on conveyor belts; the instantaneous location of moving objects is computed by the vision system acquiring images from a stationary, down looking camera. The algorithms for visual tracking of conveyor belts for moving object access are partitioned in two stages: (i) visual planning of the instantaneous destination of the robot, (ii) dynamic re-planning of the robot's destination while tracking the moving objects.
In the first method one assumes that conveyors are configured as external axes of the robot, which allows their modelling by means of a special class of variables called belt variables. A belt variable is here considered as a relative transformation (having a component variable in time) defining the location of a reference frame attached to the moving belt conveyor. By composing this time variable transformation (it reflects the contents of the belt's encoder) with the time – invariant instantaneous object location (estimated by vision for each object and compensated by the encoder offset value), the motion control algorithm will operate with a periodically updated destination, hence the robot will track the object moving on the belt.
In the second method the ensemble conveyor belt-actuator-sensor is configured as a $m \leq 3$-axis Cartesian robot, leading thus to a problem of cooperation between multiple robot manipulators subject to the multitasking control of a computer. Conceptually, the problem is solved by defining a number of *user tasks* which attach two types of "robots": the n – d.o.f. manipulator responsible with grasping on-the-fly objects moving on the conveyor belt, and the $m \leq 3$-axis robot emulating the conveyor belt under vision control. These user tasks run concurrently with the internal system tasks of a multitasking robot controller, mainly responsible for trajectory generation, axis servoing and system resources management.
Both methods use the concept of Conveyor Belt Window to implement fast reaction routines in response to emergency situations. The tracking algorithms

Source: Industrial-Robotics-Theory-Modelling-Control, ISBN 3-86611-285-8, pp. 964, ARS/plV, Germany, December 2006, Edited by: Sam Cubero

also provide collision-free object grasping by modelling the gripper's fingerprints and checking at run time whether their projections on the image plane cover only background pixels.

2. Modelling conveyors with belt variables

The problem of interest consists in building up a software environment allowing a robot controller to estimate the instantaneous position of the conveyor belt on which parts are travelling. The conveyor must be equipped with a displacement measuring device, in this case an *encoder*.
There are no constraints with respect to the position and orientation of the conveyor relative to the working area of the robot; the only requirement is that the belt's motion follows a straight line within the robot's manipulability region (Schilling, 1990). The encoder data will be interpreted by the controller as the current displacement of one of its external robot axes – in this case the tracked conveyor belt.

2.1 The special class of belt variables

The mechanism which allows specifying robot motions relative to a conveyor belt consists into modelling the belt by defining a special type of location data, named *belt variables*.

Definition 5.1: A *belt variable* is a relative homogenous transformation (having a component variable in time) which defines the location of a conveniently chosen reference frame attached to the conveyor's moving belt. The assignment of a belt variable is based on the software operation

DEFBELT %belt_variable = nominal_trans,scale_factor,

where:

- *%belt_variable* is the name of the belt variable to be defined, expressed as a 6-component homogenous transformation in minimal representation of the frame's orientation (e.g. by the Euler angles yaw, pitch and roll).
- *nominal_trans* represents the value in R^6 of the relative transformation defining the position and the orientation of the conveyor belt. The X axis of *nominal_trans* indicates the direction of motion of the belt, the XY plane defined by this transformation is parallel to the conveyor's belt surface, and the position (X,Y,Z) specified by the transformation points to the approximate centre of the belt relative to the base frame of the robot. The origin of *nominal_trans* is chosen in the middle of the robot's working displacement over the conveyor.

- *scale_factor* is the calibrating constant specifying the ratio between the elementary displacement of the belt and one pulse of the encoder.

Using such a belt variable, it becomes possible to describe the relationship between a belt encoder and the location and speed of the reference frame (conveniently chosen with respect to the manipulability domain of the robot accessing the belt) which maintains a fixed position and orientation relative to the belt (Borangiu, 2004). The informational model of the conveyor belt is its assigned belt variable, to which additional modelling data must be specified for robot-vision compatibility:

- *window parameters*, defining the working area of the robot over the conveyor belt;
- *encoder offset*, used to adjust the origin of the belt's reference frame (e.g. relative to the moment when image acquisition is triggered).

The current orientation data in a belt variable is invariant in time, equal to that expressed by *nominal_trans*. In order to evaluate the current location updated by the same belt variable, the following real-time computation has to be performed: multiplication of a unit vector in the direction of $X_{|\text{nominal\_trans}}$ by *belt_distance* – a distance derived from the encoder's contents (periodically read by the system), and then addition of the result to the position vector of *nominal_trans*. The symbolic representation of this computation is given in equations (1) and (2):

$$XYZ_{\text{instantaneous}} = XYZ_{\text{nominal}} + belt\_distance * \text{unit\_vect}(X_{|\text{nominal\_trans}}) \tag{1}$$

$$belt\_distance = (encoder\_count - encoder\_offset) * scale\_factor \tag{2}$$

Here, *encoder_count* is the encoder's read contents and *encoder_offset* will be used to establish the instantaneous location of the belt's reference frame (x_i, y_i) relative to its nominal location (x_n, y_n). In particular, the belt's offset can be used to nullify a displacement executed by the conveyor (by setting the value of the offset to the current value of the encoder's counter). The designed algorithm for visual robot tracking of the conveyor belt accounts for variable belt offsets which are frequently changed by software operations using mathematical expressions, like that included in the following V+ syntax: SETBELT %belt_variable = expression.
When the conveyor belt position is computed by referring to its assigned belt variable, the previously defined encoder offset will be always subtracted from the current position of the belt, i.e. from the encoder's current accumulated content. In the discussed method, setting a belt offset will use the real-valued

function BELT %belt_variable,mode to effectively reset the belt's position [encoder pulses].

Example 1:
A reaction routine continuously monitors a fast digital-input interrupt line which detects the occurrence of an external event of the type: "an object has completely entered the Conveyor Belt Window – and hence is completely visible". This event is detected by a photocell, and will determine an image acquisition of the moving object. The very moment the detected object is recognised as an object of interest and successfully located, the position of the belt is reset and consequently the belt variable will encapsulate from now on the current displacement of the belt relative to the belt's position in which the object has been successfully located. The V+ code is given below:

```
trigger = SIG(1001)          ;signal from photocell
save = PRIORITY  ;current priority of the robot-vision task
snap = 0                     ;reset event detection after image acquisition
success = 0           ;reset indication of successful part location
REACT –trigger,acquisition(snap,success,$name,belt_offset)
TIMER 1 = 0                  ;reset "timeout"-valued timer
IF TIMER(1)>timeout AND NOT snap  THEN
       GOTO l_end            ;no incoming parts, exit the task
 ELSE
LOCK PRIORITY + 2       ;raise priority to lock out any signals from the
          ;photocell until the current object is treated
IF success == 1  THEN
SETBELT %belt = belt_offset
IF $name == "PART"  THEN     ;if the object is of interest
Tracking the belt such that the robot picks on-the-fly the object (modelled
with the name "part") which was successfully located by vision in vis.loc
...
END
LOCK save ;re activate the REACT mechanism to check for on-off
;for on-off signal #1001 transitions
END
```

The interruption routine, automatically called by the REACT mechanism, has the form:

```
.PROGRAM acquisition(snap,success,$name,belt_offset)
 VPICTURE (cam) –1,1    ;image acquisition and recognition of one object
 snap = 1
;Locate any type of recognised object, return its name in the string
```

```
;var. $name and its location relative to the vision frame in vis_loc
 VLOCATE (cam,0) $name,vis.loc
 success = VFEATURE(1)         ;evaluate the success of the locating op.
 belt_offset = VFEATURE(8)
;The real-valued function VFEATURE(8) returns the contents of the
;belt's encoder when the strobe light for image acquisition was
;triggered.
 RETURN
.END
```

In what concerns the encoder's scale factor represented by the constant parameter *scale_factor*, it can be evaluated:

- either theoretically, knowing the mechanical coupling belt-encoder,
- or experimentally by reading the encoder's contents twice, each time when the image acquisition is triggered for a circular disk the presence of which is detected by the belt's photocell. The distance at which travel the two identical disks on the conveyor belt has been upstream set at a convenient, known value (see Fig. 1).

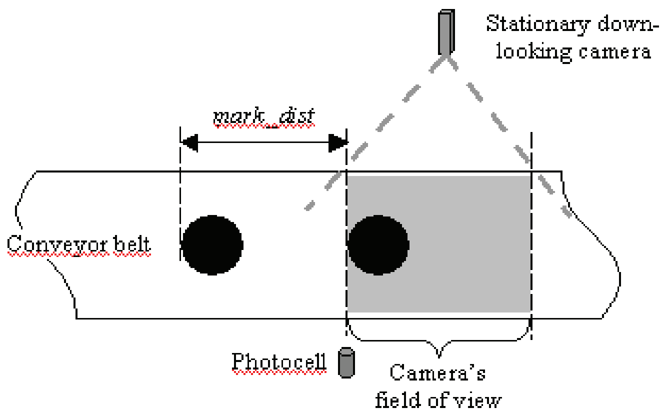


Figure 1. The experimental setup for conveyor belt calibration

2.2 The logical mechanism "Conveyor Belt Window" and emergency routines

There has been designed a logical mechanism called Conveyor Belt Window (CBW) which allows the user to check the area of the belt in which the robot will operate. A CBW defines a segment of the belt delimitated by two planes perpendicular to the direction of motion of the belt (this window is restricted only in the direction of motion of the conveyor's belt) (Borangiu & Kopacek, 2004).

Because the conveyor is modelled by a belt variable (e.g. *%belt_var*), in order to define a CBW it is necessary to refer the same belt variable and to specify two composed transformations which, together with the direction of motion of the belt, restrict the motion of the robot along a desired area of the conveyor:

WINDOW %belt_var = downstr_lim,upstr_lim,program_name,priority,

where:

- *downstr_lim* and *upstr_lim* are respectively the relative transformations defining the downstream and upstream edges of an invariant window positioned along the belt within the working space of the robot *and* the image field of the camera (it is necessary that the robot tracks and picks parts within these two limits);
- *program_name* indicates the reaction routine to be automatically called, whenever a window violation occurs while the robot tracks the conveyor belt;
- *priority* is the level of priority granted to the reaction routine. Normally, it must be greater than that of the conveyor tracking program, so that the motion of the robot can be immediately interrupted in case of window violation.

The CBW will be used not only in the stage of robot motion planning, but also at run time, during motion execution and tracking control, in order to check if the motion reference (the destination) is within the two imposed limits:

- When *a robot movement is planned*, the destination of the movement is checked against the operating CBW; if a window violation is detected, the reaction program is ignored and an error message will be issued.
- When *a robot movement relative to the conveyor belt is executed*, the destination is compared every 8 milliseconds with the window's limits; if a window violation is detected, the reaction program is automatically invoked according to its priority level and the robot will be instructed to stop tracking the belt.

There have been designed two useful CBW functions which allow the dynamic reconfiguring of programs, decisions, branching and loops during the execution of robot – vision conveyor tracking programs, function of the current value of the part-picking transformation relative to the belt, and of the current status of the belt tracking process. These functions are further introduced.
The function WINTEST(robot_transformation,time,mode) returns a value in millimetres indicating where is situated the location specified by the belt-relative composed transformation *robot_transformation*, with respect to the fixed window limits *downstr_lim* and *upstr_lim* at *time* seconds in the future, computed according to its current position and belt speed.

Finally, the argument *mode* is a real-valued expression which specifies whether the result of the WINTEST function represents a distance inside or outside the predefined conveyor belt window. For example, if *mode* is positive, the value returned by WINTEST will be interpreted as:

0: the composed, belt-relative location is inside the CBW;
<0: the location is upstream of *upstr_lim* of the CBW;
>0, the location is downstream of the *dwnstr_lim* of the CBW.

Hence, the returned value conforms to the WINDOW model shown in Fig. 2, for which the value returned by the function WINDOW increases as the belt-relative location moves downstream.

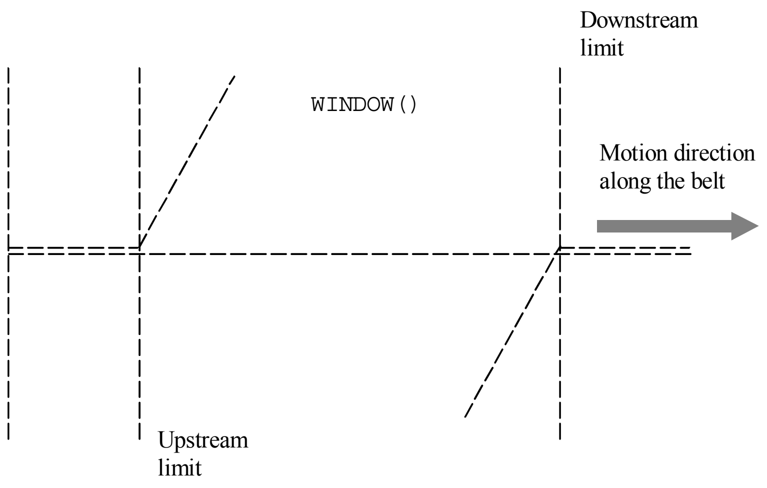


Figure 2. The WINDOW function for *mode* > 0

For robots tracking conveyor belts in order to pick moving objects recognised and located by vision, the belt-relative transformation is *%belt_var:part.loc* (variable in time), obtained by the composition of:

- *%belt_var*, models the conveyor belt,
- *part.loc*, is a composed, *time invariant* transformation expressing the gripper frame (x_g, y_g, z_g) relative to the base frame of the robot (x_0, y_0, z_0) at the moment the object was identified and located by vision.

For example, the distance WINTEST(%belt:part.loc,4,1) is positive if, in 4 seconds from the time being, the belt-relative part picking location will be outside the window defined for the conveyor belt modelled by *%belt*.
If the robot tries to move towards a belt-relative location that has not yet appeared inside the belt window (it is still upstream relative to the CBW), the motion control algorithm has been designed with two options:

- *temporarily stops the robot*, delaying thus the motion planning, until the time-variable destination enters the belt window;
- *definitively stops the robot* and generates immediately an error message.

Also, the control algorithm generates a condition of window violation anytime the vision-based robot motion planner computes a destination which is downstream the CBW, or will exit the CBW at the predicted time the robot will reach it. The function BELTSTATUS indicates the current status of the belt tracking process: *robot tracking the belt*; *destination upstream*; *destination downstream*; *window violation*, real-time information which can be used to dynamically reconfigure the robot – vision task.

2.3 Robot locations, frames and belt-relative movements planned by vision

To command the belt-relative motion of a robot with linear interpolation in the Cartesian space, i.e. to define an end-tip transformation relative to an instantaneous location of a moving frame (x_i, y_i) on the conveyor belt, the already defined belt variable (which models the conveyor belt as a relative transformation having time variable components along the *X* (and possibly *Y*) Cartesian axes) will be composed with the time-invariant end-tip transformation relative to the base of the robot (which is computed at run time by the vision part of the system).
The result will be a time-variable transformation updating the position reference for the robot. This reference or target destination tracks an object moving on the belt, to be picked by the robot's gripper. The target destination is:

- *planned once* at runtime by vision, as soon as the object is perfectly visible to the camera, either inside the manipulability area of the robot or upstream this area;

- *updated every 8 milliseconds* by the motion controller based on the current position data read from the belt's encoder, until the robot's end-point completes the necessary percentage of its motion segment towards the part's grasping location.

The research on Guidance Vision for Robots (GVR) accessing moving targets was directed to develop a *convergent* motion control algorithm for visually plan the motion of the robot as a result of object detection, recognition and locating on a moving conveyor belt, and than track the object in order to grasp it inside a conveniently defined belt window. The main idea relies on dynamically changing the visually computed destination of the robot end point by composing it with a belt-related transformation updated every 8 milliseconds from the encoder data.

If a stationary camera looks down at the conveyor belt, and supposing that its field of view covers completely a conveyor belt window defined inside the working area of the robot (after execution of a camera – robot calibration session), then the image plane can be referred by the time – invariant frame (x_{vis}, y_{vis}) as represented in Fig. 3.

It is also assumed that the *X* axes of the reference frame of the robot (x_0), of the conveyor's direction of motion (x_n) and of the image plane (x_{vis}) are parallel. The conveyor belt is modelled by the belt variable %belt. Parts are circulating on the belt randomly; their *succession* (current part type entering the CBW), *distance from the central axis of the conveyor* and *orientation* are unknown. The "Look-and-Move" interlaced operating principle of the image processing section and motion control section is used (Hutchinson, 1996), (Borangiu, 2001), (West, 2001), (Adept, 2001). According to this principle, while an image of the CBW is acquired and processed for object identification and locating, no motion command is issued and reciprocally, the camera will not snap images while the robot tracks a previously located part in order to pick it "on-the-fly".

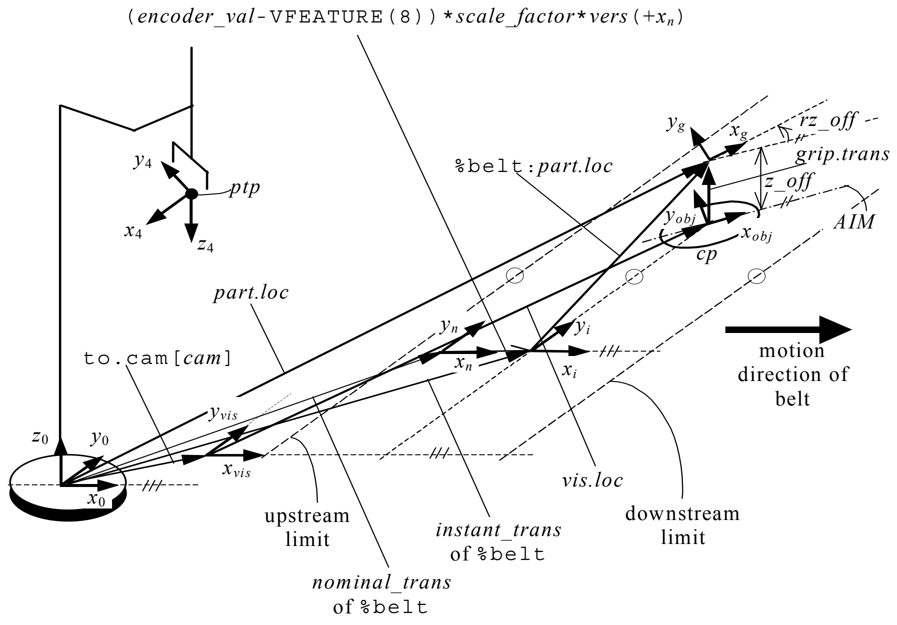


Figure 3. Robot-Vision and belt-relative transformations for conveyor tracking

The robot motion control algorithm for tracking the conveyor belt in order to pick "on-the-fly" one by one objects recognised and located by vision computing consists of the following basic steps:

1. *Triggering the strobe light* (synch./asynch. relative to the read cycle of the video camera) when *image acquisition* is requested from a fast digital-input interrupt line connected to a photocell mounted at the upstream limit of the CBW. The interrupt line signals that an object has completely entered the belt window.
2. *Recognizing a single object* that just completely entered the belt window. Object recognition is exclusively based in this approach on the match with previously learned models of all objects of interest (Lindenbaum, 1997).
3. *Locating the object* which was recognised, by computing the coordinates of its centre of mass and the angle between its minimum inertia axis (MIA) and x_{vis}. As can be seen in Fig. 3, the object-attached frame (x_{obj}, y_{obj}) has the abscissa aligned with the minimum inertia axis (MIA), and the current location of the object in the image plane is computed by the vision section and returned in *vis.loc*.
4. *Planning the instantaneous destination of the robot*. Once the object is *recognized* as the instance of a model and *located*, the related grasping transformation *grip.trans* is called. Assuming that the grasping style is such that the projection of the gripper's centre on the image plane coincides with the object's centre of mass, the gripper-attached frame (x_g, y_g) will be offset relative to the object-attached frame along z_0 by *z_off* millimetres and turned with *r_off* degrees about z_0. Now, using the relative transformation to.cam[*cam*](as output of the camera-robot calibration session) relating the vision frame $(x_{vis}, y_{vis}, z_{vis})$ to the base frame of the robot (x_0, y_0, z_0), the current destination of the robot (for a frozen conveyor belt) is computed from the vision data as a composed transformation *part.loc*, expressing the gripper frame relative to the robot base frame:

   part.loc = to.cam[cam]:vis.loc:grip.trans

5. *Synchronising the encoder belt with the motion of the object* recognized in the belt window. This operation consists into setting the offset of the conveyor belt at a correct value. The operation

   SETBELT %belt = encoder_val(strobe)

establishes the point of interest of the conveyor belt modelled with %belt as the point corresponding to the current value *encoder_val*(*strobe*) of the encoder counter at the time the strobe light was triggered. This value is available immediately after object locating. Thus, as soon as one object is recognized and located, the current belt position, identified by (x_i, y_i), will be reset since:

$$\begin{aligned} xyz_i = xyz_n &+ (encoder\_val - encoder\_val(strobe)) * \\ &* scale\_factor * \mathrm{unit\_vect}(x_n) = xyz_n \end{aligned} \tag{3}$$

6. *Tracking and picking the object moving on the belt.* This requires issuing a linear motion command in the Cartesian space, relative to the belt. A composed relative transformation %belt:part.loc, expressing the current computed location of the gripper relative to the instantaneous moving frame (x_i, y_i), is defined. Practically, the tracking procedure begins immediately after the instantaneous position of the belt – expressed by the frame (x_i, y_i) has been initialized by the SETBELT operation, and consists into periodically updating the destination of the gripper by shifting it along the x_n axis with encoder counts accumulated during successive sampling periods $\ldots, t_{k-1}, t_k, t_{k+1}, \ldots \quad \Delta t = t_{k+1} - t_k = \text{const}:$

$$\begin{aligned} \%belt : part.loc\big|_{t_{k+1}} = \mathtt{SHIFT}\ (\%belt : part.loc\big|_{t_k}\ \mathtt{BY}\ldots \\ \ldots\ encoder\_count(t_{k+1} - t_k) * scale\_factor, 0, 0) \end{aligned} \tag{4}$$

```
MOVES %belt:part.loc          ;go towards the moving target
CLOSEI                        ;grasp "on the fly" the object
```

7. Once the robot commanded towards a destination relative to the belt, the gripper will continuously track the belt until a new command will be issued to approach a location which is not relative to the belt.

For belt-relative motions, the destination changes continuously; depending on the magnitude and the variations of the conveyor speed it is possible that the robot will not be able to attain the final positions within the default error tolerance.

In such cases, the error tolerance must be augmented. In extreme cases it will be even necessary to totally deactivate the test of final error tolerance. Fig. 4

presents the designed robot motion control algorithm for tracking the conveyor belt in order to pick "on-the-fly" an object recognized and located by vision computation inside the a priori defined belt window. A REACT mechanism continuously monitors the fast digital-input interrupt line which signals that an object has completely entered the belt window. The robot motion relative to the belt will be terminated:

- when moving the robot towards a *non belt-relative location* or
- when a *window violation* occurs.

***Example 2*:**
The following series of instructions will move the robot end-effector towards a belt-relative location part_2 (the belt is modelled as %belt[1]), open the gripper, track the conveyor belt for 5 seconds (in fact the location part_2 on the belt), close the gripper and finally leave the belt to move towards a fixed location.

```
MOVES %belt[1]:part_2
OPENI
DELAY 5.0
CLOSEI
MOVES fixed_location
```

When defining the Conveyor Belt Window, a special high-priority routine can be specified, which will be automatically invoked to correct any window violation during the process of tracking moving objects. In such situations the robot will be accelerated (if possible) and the downstream limit temporarily shifted in the direction of motion of the conveyor belt (within a timeout depending on the belt's speed) in which the error tolerance must be reached (Espiau, 1992), (Borangiu, 2002).

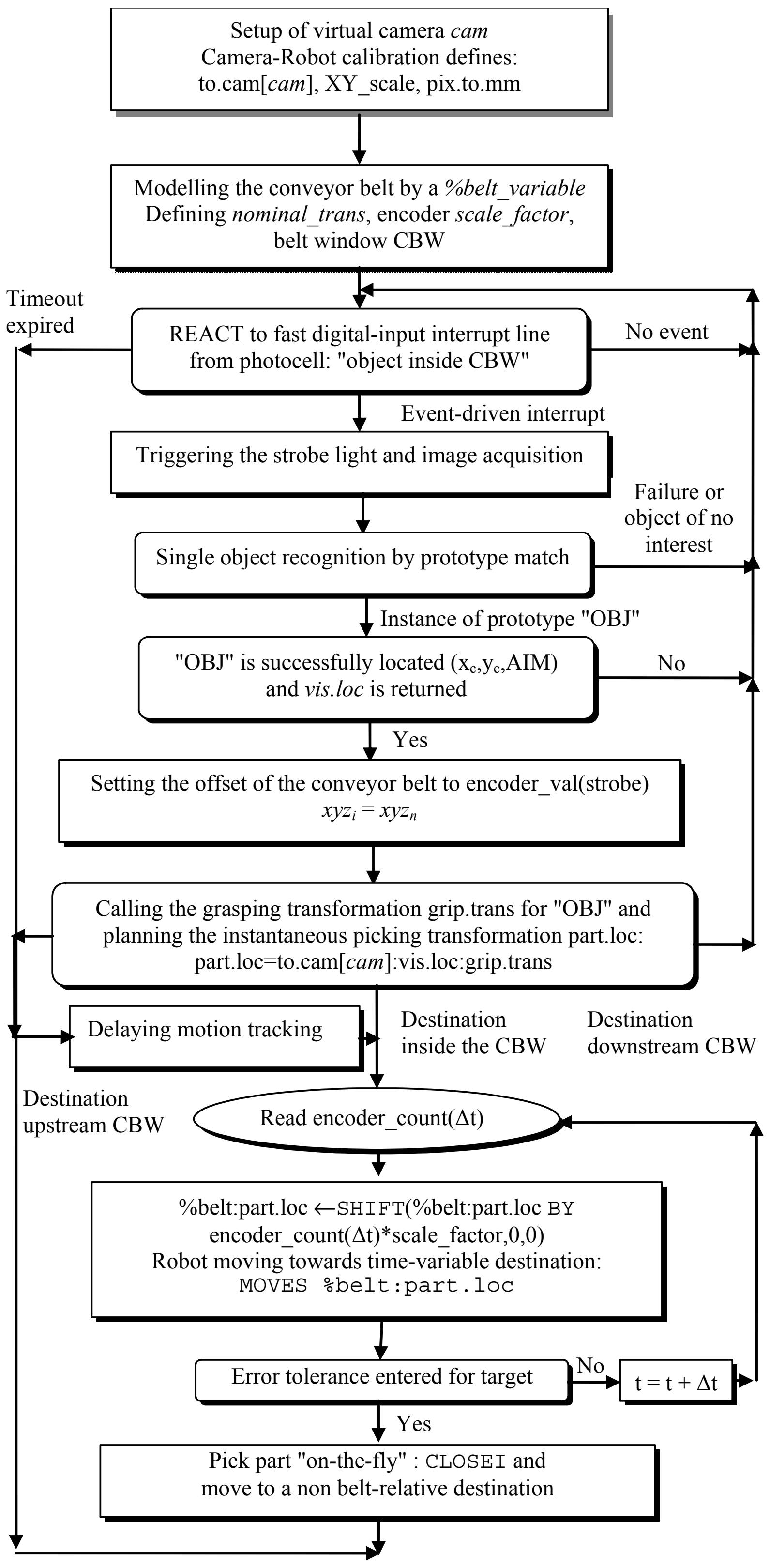

Setup of virtual camera *cam*
Camera-Robot calibration defines:
to.cam[*cam*], XY_scale, pix.to.mm
Modelling the conveyor belt by a *%belt_variable*
Defining *nominal_trans*, encoder *scale_factor*,
belt window CBW
Timeout expired
REACT to fast digital-input interrupt line from photocell: "object inside CBW"
No event
Event-driven interrupt
Triggering the strobe light and image acquisition
Failure or object of no interest
Single object recognition by prototype match
Instance of prototype "OBJ"
"OBJ" is successfully located (x_c,y_c,AIM) and *vis.loc* is returned
No
Yes
Setting the offset of the conveyor belt to encoder_val(strobe)
$xyz_i = xyz_n$
Calling the grasping transformation grip.trans for "OBJ" and planning the instantaneous picking transformation part.loc:
part.loc=to.cam[*cam*]:vis.loc:grip.trans
Delaying motion tracking
Destination inside the CBW
Destination downstream CBW
Destination upstream CBW
Read encoder_count(Δt)
%belt:part.loc ←SHIFT(%belt:part.loc BY encoder_count(Δt)*scale_factor,0,0)
Robot moving towards time-variable destination:
MOVES %belt:part.loc
Error tolerance entered for target
No
t = t + Δt
Yes
Pick part "on-the-fly" : CLOSEI and move to a non belt-relative destination

Figure 4. The robot motion algorithm for visual tracking of the conveyor belt

3. Tracking conveyors as *m*≤3 Cartesian axis robots

According to the second tracking method, the ensemble conveyor belt + actuator + sensor is configured as an $m \leq 3$*-axis Cartesian robot*, which leads to a problem of cooperation between multiple robots subject to multitasking computer control. The V+ structured programming environment is used for exemplifying the multi tasking control of robots visually tracking moving objects on multiple belts.

3.1 Multitasking control for robot cooperation

Conceptually, the problem is solved by defining a number of *user tasks* which attach two types of "robots": the n – d.o.f. manipulator responsible with grasping on-the-fly objects moving on the conveyor belt, and the $m \leq 3$-axis robot emulating the conveyor belt under vision control. These user tasks run concurrently with the internal system tasks of a multitasking robot controller, mainly responsible for trajectory generation, axis servoing and system resource management (Adept, 2001).

In this respect, there are three tasks to be at least defined for the tracking problem:

1. Task 1: Dynamic re-planning the destination location (grasping the moving object) for the robot manipulator.
2. Task 2: Continuously moving (driving) the $m \leq 3$-axis vision belt. In the most general case, the belt moves along any 3D-direction relative to the robot base frame (x_0, y_0, z_0).
3. Task 3: Reading *once* the belt's location the very moment an object of interest has been recognised, located and its grasping estimated as collision-free, and then *continuously* until the object is effectively picked.

3.1.1 Specifying tasks, time slices and priorities

A multitasking robot control system appears to execute all these program tasks at the same time. However, this is actually achieved by rapidly switching between the tasks many times each second, each task receiving a fraction of the total time available. This is referred to as concurrent execution (Zhuang, 1992), (Borangiu, 2005).

The amount of time a particular program task receives is caused by two parameters: its *assignment* to the various time slices, and its *priority* within the time slice. One assumes that, in the multitasking operating system, each *system cycle* is divided into 16 *time slices* of one millisecond each, the slices being numbered 0 through 15. A single occurrence of all 16 time slices is referred to

as a *major cycle*. For a robot each of these cycles corresponds to one output from the trajectory generator to the servos.
A number of seven user tasks, e.g. from 0 to 6, will be used and their configuration tailored to suit the needs of specific applications. Each program task configured for use requires dedicated memory, which is not available to user programs. Therefore, the number of tasks available should be made no larger than necessary, especially if memory space for user programs is critical.
When application programs are executed, their program tasks are normally assigned default time slices and priorities according to the current system configuration. The defaults can be overridden temporarily for any user program task, by specifying the desired time slice and priority parameters of the EXECUTE initiating command.
Tasks are scheduled to run with a specified priority in one or more time slices. Tasks may have priorities from –1 to 64, and the priorities may be different in each time slice. The priority meanings are:

–1	Do not run in this slice even if no other task is ready to run.
0	Do not run in this slice unless no other task from this slice is ready to run.
1-64	Run in this slice according to specified priority. Higher priority tasks may lock lower ones. Priorities are broken into the following ranges:
1-31	Normal user task priorities;
32-62	Used by robot controller's *device drivers* and *system tasks;*
63	Used by *trajectory generator*. Do not use 63 unless you have very short task execution times, because use of these priorities may cause jerks in the robot trajectories;
64	Used by the *servo*. Do not use 64 unless you have very short task execution times, because use of these priorities may cause jerks in the robot trajectories.

The V+ operating system has a number of *internal* (system) *tasks* that compete with *application* (user) *program tasks* for time within each time slice:

- On motion systems, the V+ *trajectory generator* runs (at the highest priority task) in slice #0 and continues through as many time slices as necessary to compute the next motion device set point.

- On motion systems, the CPU running servo code runs the *servo task* (at interrupt level) every 1 or 2 milliseconds (according to the controller configuration utility).

The remaining time is allocated to user tasks, by using the controller configuration utility. For each time slice, you specify which tasks may run in the slice and what priority each task has in that slice.

3.1.2 Scheduling of program execution tasks

Vision guided robot planning ("object recognition and locating"), and dynamical re-planning of robot destination ("robot tracking the belt") should always be configured on user tasks 0 or 1 in "Look-and-Move" interlaced robot motion control, due to the continuous, high priority assignment of these two tasks, over the first 13 time slices. However, vision guidance and motion re-planning programs complete their computation in less than the 13 time slices (0-12).
Consequently, in order to give the chance to conveyor-associated tasks ("drive" the vision belt, "read" the current position of the vision belt") to provide the "robot tracking" programs with the necessary position update information earlier than the slice 13, and to the high-priority trajectory generation system task to effectively use this updates, a WAIT instruction should be inserted in the loop-type vision guidance and motion re-planning programs of tasks 0 and/or 1.
A WAIT *condition* instruction with no argument will suspend then, once per loop execution, the motion re-planning program, executing on user task 1, until the start of the next major cycle (slice 0). At that time, the "vision processing and belt tracking" task becomes runnable and will execute, due to its high priority assignment.
Due to their reduced amount of computation, programs related to the management of the conveyor belt should be always assigned to tasks 2, 3, 5 or 6 if the default priority scheme is maintained for user program tasks, leaving tasks 1 and 2 for the intensive computational vision and robot motion control.
Whenever the current task becomes inactive, the multitasking OS searches for a new task to run. The search begins with the highest priority task in the current time slice and proceeds through in order of descending priority. If multiple programs wait to run in the task, they are run according to relative program priorities. If a runnable task is not found, the next higher slice is checked. All time slices are checked, wrapping around from slice 15 to slice 0 until the original slice is reached. Whenever a 1 ms interval expires, the system performs a similar search of the next time slice; if this one does not contain a runnable task, the currently executing task continues.
If more than one task in the same time slice has the same priority, they become part of a *round-robin scheduling group*. Whenever a member of a round-robin group is selected by the normal slice searching, the group is scanned to find the member of the group that run most recently. The member that follows the most recent is run instead of the one which was originally selected.

The V+ RELEASE program instruction may be used to bypass the normal scheduling process by explicitly passing control to another task. That task then goes to run in the current time slice until it is rescheduled by the 1 ms clock. A task may also RELEASE to *anyone*, which means that a normal scan is made of all other tasks to find one that is ready to run. During this scan, members of the original task's round-robin group (if any) are ignored. Therefore, a RELEASE to anyone cannot be used to pass control to a different member of the current group.
Round-robin groups are treated as a single task. If any member of the group is selected during the scan, then the group is selected. The group is scanned to find the task in the group following the one which ran most recently, and that task is run. Within each time slice, the task with highest priority can be locked out only by a servo interrupt. Tasks with lower priority, defined for driving the conveyor belt and reading position data from its encoder, can run only if the higher-priority task, defined for vision guidance of the *n*–d.o.f. robot and for tracking the 1–d.o.f. robot-like conveyor belt, is inactive or waiting. A user task waits whenever:

- The program issues an input or an output request that causes a wait.
- The program executes a robot motion instruction while the robot is still moving in response to a previous motion instruction.
- The program executes a WAIT or WAIT.EVENT program instruction.

If a program is executing continuously without performing any of the above operations, it locks out any lower-priority tasks in its time slice. Thus, programs that execute in continuous loops, like vision guidance and motion re-planning for belt tracking, should generally execute a WAIT (or WAIT.EVENT) instruction occasionally (for example, *once each time through the loop*).
If a program potentially has a lot of *critical processing* to perform, its task should be in *multiple slices*, and the task should have the *highest priority* in these slices. This will guarantee the task's getting all the time needed in the multiple slices, plus (if needed) additional unused time in the major cycle.

Fig. 5 shows the *task scheduler algorithm* which was designed for an *n*-d.o.f. robot tracking a continuously updated object grasping location, and picking the object "on-the-fly" from a conveyor belt, when motion completes. The object is recognized and located by vision, and updating of its position is provided by encoder data reads from the conveyor belt modelled as a 1-d.o.f. robot. The priority analysis and round-robin member selection are also indicated.
The problem of conveyor tracking with vision guiding for part identification and locating required definition of three user tasks, to which programs were associated:

1. Task 1: program "**track**" executes in this task, with robot 1 (4-d.o.f. SCARA) selected. This program has two main functions, carried out in a 2 – *stage* sequence:

   STAGE 1: Continuous *checking* whether an object travelling on the *vision belt* entered the field of view of the camera and the reachable workspace of the SCARA robot. If such an event occurs, the vision is activated to *identify* whether the object is of interest and to *locate* it. Processing on this stage terminates with the *computation of the end-effector's location* which would move the robot in the object picking location evaluated once by vision, according to a predefined grasping style, if the belt were stopped.

   STAGE 2: Continuously *re-planning* the end-effector's location, computed once by vision, by *consuming the belt position data* produced by encoder reads in program "**read**" which executes on task 3, and by *dynamically altering the robot's target* in the current motion segment.

2. Task 2: program "**drive**" executes in this task, with robot 2 (the 1-d.o.f. conveyor belt) selected. This program *moves the belt* in linear displacement increments, at a sufficiently high rate to provide a jerk-free, continuous belt motion. This program executes in both stages of the application, previously defined.

3. Task 3: program "read" executes in this task, with robot 2 selected. This program executes differently in the two stages of the application:

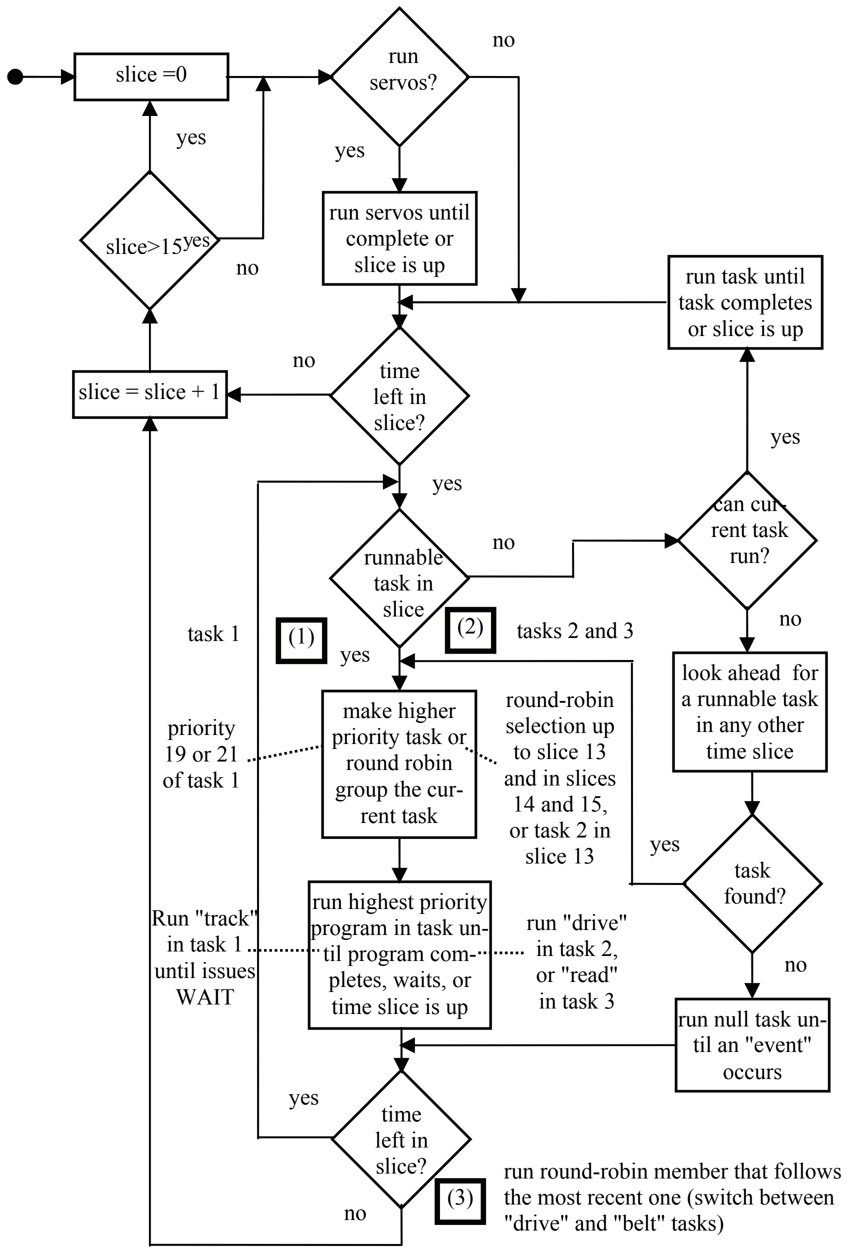

slice =0
run servos?
no
yes
run servos until complete or slice is up
slice>15
yes
yes
no
run task until task completes or slice is up
no
time left in slice?
slice = slice + 1
yes
yes
runnable task in slice
no
can current task run?
task 1
(1)
(2)
tasks 2 and 3
yes
no
priority 19 or 21 of task 1
make higher priority task or round robin group the current task
round-robin selection up to slice 13 and in slices 14 and 15, or task 2 in slice 13
look ahead for a runnable task in any other time slice
yes
task found?
Run "track" in task 1 until issues WAIT
run highest priority program in task until program completes, waits, or time slice is up
run "drive" in task 2, or "read" in task 3
no
run null task until an "event" occurs
yes
time left in slice?
no
(3)
run round-robin member that follows the most recent one (switch between "drive" and "belt" tasks)

Figure 5. Task scheduler for multitasking robot control and conveyor tracking

STAGE 1: Executes *a single time* upon receiving an input signal from vision in task 1, confirming the recognition and locating of a part. In response, "**drive**" *reads the instantaneous belt position*, which from now on will be used as an offset for the position updates.

STAGE 2: Continuously *reads the belt position,* upon a request issued by "**track**" in task 1, when it starts its dynamic target re planning process.

From the three user tasks, the default priority assignment is maintained. This leads to the following priority analysis for a major cycle:

- Task 1 has the highest priority in time slices 0 – 12 (inclusively), with values of 19, 21, 9 and 11.
- Task 2 has the highest priority (20) in a single time slice: 13.
- Task 3 never detains explicitly a position of highest priority with respect to tasks 1 and 2.

The three tasks become part of a round-robin group as follows: tasks 2 and 3 in slices 0 – 12 inclusively; tasks 1, 2 and 3 in slices 14 and 15. Because tasks 2 and 3 are in more than one round-robin group on different slices, then all three tasks in the corresponding pairs of different slices appear to be in a big group. As a result of the priority scan and scheduling, the programs in the three user tasks execute as follows:

<u>STAGE 1</u> – vision is processing, the robot is not moving and no WAIT is issued by task 1 (Fig. 6):

– Task 1 runs: in slices 0 – 12 (it detains the highest priority), in slice 14 (it is member of the round-robin group following task 2 that run more recently – in slice 13) only *before generating the request* for task 3 to compute the instantaneous offset belt position when vision located the object, and in slice 15 only *after generating this request* (it is member of the round-robin group following task 3 that run more recently – in slice 14).

– Task 2 runs in slice 13 (it detains the highest priority), and in slice 15 (it is member of the round-robin group following task 1 that run more recently – in slice 14) only *before* task 1 *generates the request* for task 3 to compute the instantaneous offset belt position when vision located the object.

– Task 3 runs in slice 14 (it is member of the round-robin group following task 2 that run more recently – in slice 13) only *after receiving the request* from task 1 to compute the instantaneous offset belt position when vision located the object.

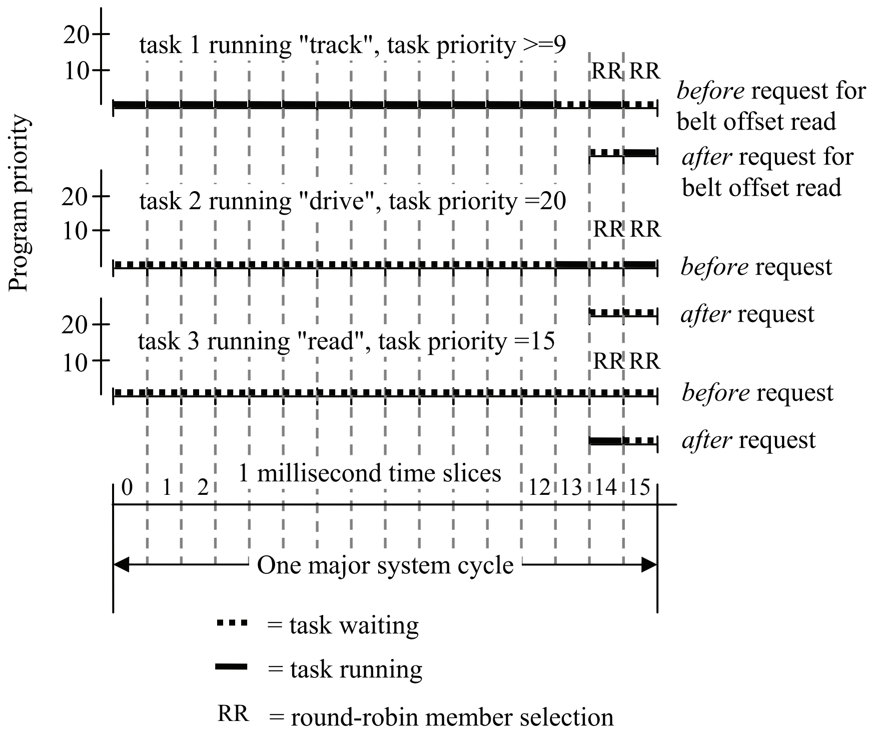

Program priority
20
10
task 1 running "track", task priority >=9
RR RR
before request for belt offset read
after request for belt offset read
task 2 running "drive", task priority =20
before request
after request
task 3 running "read", task priority =15
before request
after request
0 1 2 1 millisecond time slices 12 13 14 15
One major system cycle
= task waiting
= task running
RR = round-robin member selection

Figure 6. Priority assignment and tasks running in STAGE 1 of vision guidance for robot motion planning

<u>STAGE 2</u> – *vision is not processing, the* SCARA *robot is moving and* WAIT *commands are issued in* task 1 *by the* "**track**" *program after each re-planning of the end-effector's target destination within one major cycle of 16 milliseconds* (Fig. 7):

– Task 1 runs in slices $i-j,\ i \le j, i \ge 0, j \le 12,$ (when it detains the highest priority), i.e. starting with the moment when it is authorized to run by the highest-priority system tasks "**trajectory generation**" and "**servo**" (in slice *i*), and executing until it *accesses the position update* provided by task 3 *from the most recent belt encoder read, alters the last computed end-effector destination* and issues a WAIT (in slice *j*), to give the trajectory generator a chance to execute. From this time moment, task 1 becomes inactive for the rest of the 16 milliseconds of the current major cycle, until slice 0 of the next system cycle when it waits to be selected by the scheduler and authorized to run.

– Task 2 runs: in slices $(j+1)-12$ switching alternatively with task 3 whenever it is selected as the member of the round-robin group following task 3 that run most recently, in slice 13 (it detains the highest priority), and in slice 15 (it is member of the round-robin group following task 3 that run more recently – in slice 14). Task 2 runs always exactly for

1 millisecond whenever selected, so that the round-robin group scanning authorises task 3 to run always at the beginning of the next time slice.

– Task 3 runs in slices $(j+1)-12$ switching alternatively with task 2 whenever it is selected as the member of the round-robin group following task 2 that run most recently, and in slice 14 (it is member of the round-robin group following task 2 that run more recently – in slice 13). The task 3 runs, whenever selected, for less than 1 millisecond and issues a RELEASE "to anyone" in the current slice, allowing selection of task 2 (in the same round-robin group) for running exactly at the beginning of the next time slice.

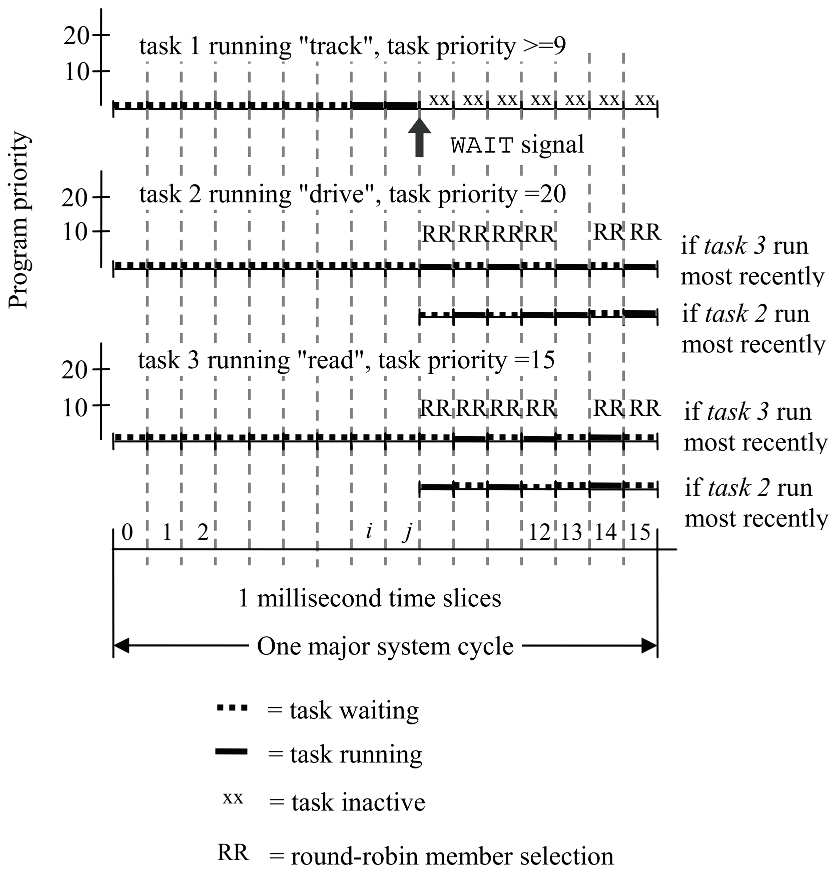

Program priority
20
10
task 1 running "track", task priority >=9
xx
WAIT signal
task 2 running "drive", task priority =20
RR
if task 3 run most recently
if task 2 run most recently
task 3 running "read", task priority =15
0 1 2 i j 12 13 14 15
1 millisecond time slices
One major system cycle
= task waiting
= task running
xx = task inactive
RR = round-robin member selection

Figure 7. Priority assignment and tasks running in STAGE 2 of robot motion re-plan

3.2 Dynamically altering belt locations for collision-free object picking on-the-fly

The three previously discussed user tasks, when runnable and selected by the system's task scheduler, attach respectively the robots:

- Task 1: **robot 1** – a SCARA-type robot Cobra 600TT was considered;
- Task 2 and 3: **robot 2** – the "vision conveyor belt" of a flexible feeding system.

In multiple-robot systems like the one for conveyor tracking, SELECT robot operations select the robot with which the current task must communicate. The SELECT operation thus specifies which robot *receives motion instructions* (for example, DRIVE to move the *vision belt* in program "**drive**", or MOVES to move the SCARA in program "**track**") and *returns robot-related information* (for example, for the HERE function accessing the current vision belt location in program "**read**").

Program "**track**" executing in task 1 has two distinct timing aspects, which correspond to the partitioning of its related activities in STAGE 1 and STAGE 2. Thus, during STAGE 1, "**track**" waits first the occurrence of the on-off transition of the input signal generated by a photocell, indicating that an object passed over the sensor and will enter the field of view of the camera. Then, after waiting for a period of time (experimentally set up function of the belt's speed), "**track**" commands the vision system to acquire an image, identify an object of interest and locate it.

During STAGE 2, "**track**" alters continuously, once each major 16 millisecond system cycle, the target location of the end-effector – part.loc (computed by vision) by composing the following relative transformations:

```
SET part.loc = to.cam[1]:vis.loc:grip.part
```

where grip.part is the learned grasping transformation for the class of objects of interest. The updating of the end-effector target location for grasping one moving object uses the command ALTER()Dx,Dy,Dz,Rx,Ry,Rz, which specifies the magnitude of the real-time path modification that is to be applied to the robot path during the next trajectory computation. This operation is executed by "**track**" in task 1 that is controlling the SCARA robot in *alter mode*, enabled by the ALTON command. When *alter* mode is enabled, this instruction should be executed once during each trajectory cycle. If ALTER is executed more often, only the last set of values defined during each cycle will be used. The arguments have the meaning:

- Dx,Dy,Dz: optional real values, variables or expressions that define the translations respectively along the X, Y, Z axes;

- Rx,Ry,Rz: optional real values, variables or expressions that define the rotations respectively about the X,Y,Z axes.

The ALTON mode operation *enables real-time path-modification mode* (alter mode), and specifies the way in which ALTER coordinate information will be interpreted. The value of the argument mode is interpreted as a sequence of two bit flags:

Bit 1 (LSB): If this bit is set, coordinate values specified by subsequent ALTER instructions are interpreted as incremental and are accumulated. If *this bit is clear*, each set of coordinate values is interpreted as the *tota*l (non cumulative) *correction to be applied*. The program "**read**" executing in task 3 provides at each major cycle the updated position information y_off of the robot 2 – the vision belt along its (unique) Y motion axis, by subtracting from the current contents *pos* the belt's offset position *offset* at the time the object was located by vision: y_off = *pos* – *offset*. The SCARA's target location will be altered therefore, in non cumulative mode, with y_off.

Bit 2 (MSB): If *this bit is set*, coordinate values specified by the subsequent ALTER instructions are interpreted to be *in the World coordinate system*, to be preferred for belt tracking problems.

It is assumed that the axis of the vision belt is parallel to the Y_0 robot axis in its base. Also, it is considered that, following the belt calibrating procedure described in Section 2.1, the coefficient *pulse.to.mm*, expressing the ratio between one belt encoder pulse and one millimetre, is known.
The repeated updating of the end-effector location by altering the part.loc object-grasping location proceeds in task 1 by "**track**" execution, until motion stops at the (dynamically re-) planned grasping location, when the object will be picked-on-the-fly (Borangiu, 2006). This stopping decision is taken by "**track**" by using the STATE (select) function, which returns information about the state of the robot 1 selected by the task 1 executing the ALTER loop. The argument select defines the category of state information returned. For the present tracking software, the data interpreted is "Motion stopped at planned location", as in the example below:

Example 3:

The next example shows how the STATE function is used to stop the continuous updating of the end-effector's target location by altering every major cycle the position along the Y axis. The altering loop will be exit *when motion stopped at planned location*, i.e. when the robot's gripper is in the desired picking position relative to the moving part.

```
ALTON () 2                        ;Enable altering mode
MOVES part.loc                    ;Robot commanded to move in grasp location
                                  ;computed by vision (VLOCATE)
WHILE STATE(2)<>2 DO              ;While the robot is far from the moving
                                  ;target (motion not completed at planned
                                  ;location
ALTER () ,-pulse.to.mm*y_off      ;Continuously alter the
                                  ;target grasping location
WAIT                              ;Wait for the next major time cycle to give the
                                  ;trajectory generator a chance to execute
END
ALTOFF                            ;Disable altering mode
CLOSEI                            ;Robot picks the tracked object
DEPARTS                           ;Robot exist the belt tracking mode
MOVES place                       ;Robot moves towards the fixed object-
                                  placing loc
```

After alter mode terminates, the robot is left at a final location that reflects both the destination of the last robot motion and the total ALTER correction that was applied.

Program "**drive**" executing in task 2 has a unique timing aspect in both STAGES 1 and 2: when activated by the main program, it issues continuously motion commands DRIVE joint,change,speed, for the individual joint number 1 of robot 2 – the vision belt (changes in position are 100 mm; several speeds were tried).

Program "**read**" executing in task 3 evaluates the current motion of robot 2 – the vision belt along its single axis, in two different timing modes. During STAGE 1, upon receiving from task 1 the info that an object was recognised, it computes the belt's offset, reads the current robot 2 location and extracts the component along the Y axis. This invariant offset component, read when the object was successfully located and the grasping authorized as collision-free, will be further used in STAGE 2 to estimate the non cumulative updates of the y_off motion, to alter the SCARA's target location along the Y axis.
The cooperation between the tasks on which run the "track", "drive" and "read" programs is shown in Fig. 8.

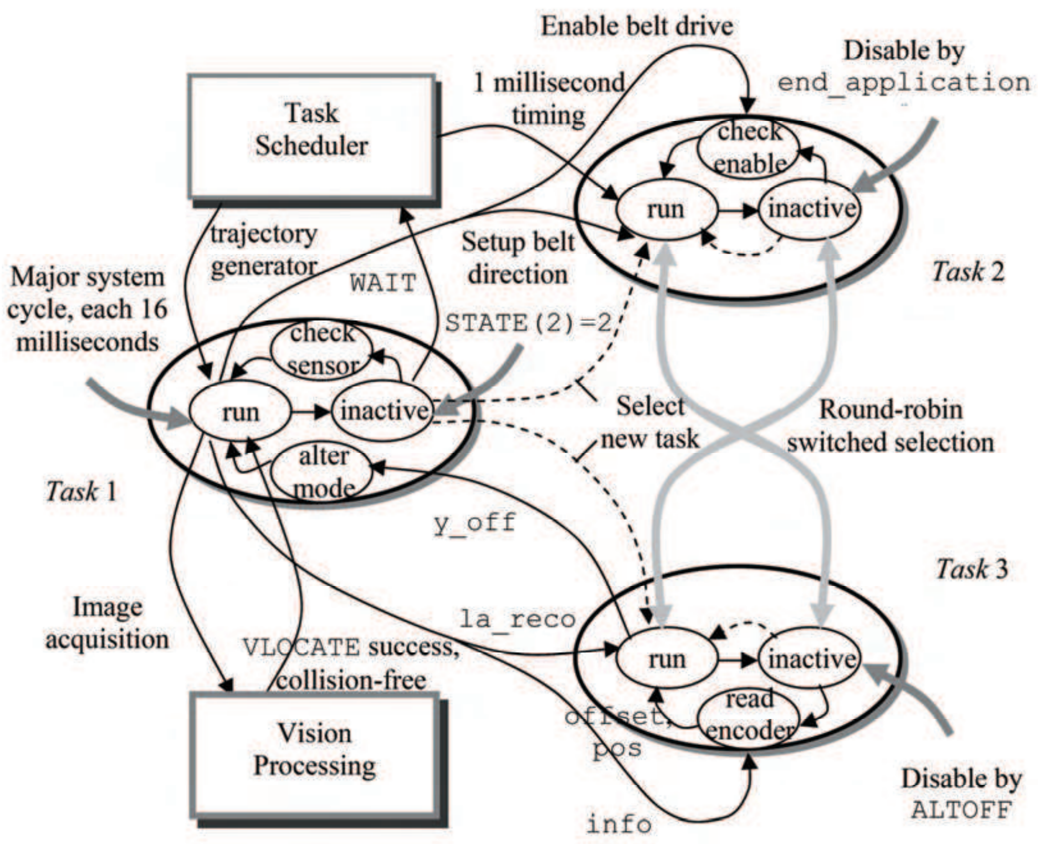


Figure 8. Cooperation between the tasks of the belt tracking control problem

4. Authorizing collision-free grasping by fingerprint models

Random scene foregrounds, as the conveyor belt, may need to be faced in robotic tasks. Depending on the parts shape and on their dimension along z^+, grasping models $\mathcal{G}s\_m$ are off line trained for prototypes representing object classes. However, if there is the slightest uncertainty about the risk of collision between the gripper and parts on the belt – touching or close one relative to the other –, then *extended* grasping models $\mathcal{EG}\_m = \{\mathcal{G}s\_m, \mathcal{FGP}\_m\}$ must be created by the adding the gripper's *fingerprint model* $\mathcal{FGP}\_m$ to effectively authorize part access only after *clear grip tests* at run time.

Definition.

A *multiple fingerprint model* $\mathcal{MFGP}\_m(\mathcal{G},\mathcal{O}) = \{\mathcal{FGP}\_m_1(\mathcal{G},\mathcal{O}),\ldots,\mathcal{FGP}\_m_k(\mathcal{G},\mathcal{O})\}$ for a p-fingered gripper $\mathcal{G}$ and a class of objects $\mathcal{O}$ describes the <u>shape</u>, <u>location</u> and <u>interpretation</u> of k sets of p projections of the gripper's fingerprints onto the image plane x_{vis}, y_{vis} for the corresponding k grasping styles $\mathcal{G}s\_m_i, i = 1,\ldots,k$ of $\mathcal{O}$-class instances. A $\mathcal{FGP}\_m_i(\mathcal{G},\mathcal{O})$ model has the following parameter structure:

- $finger\_shape(\mathcal{G}) = number, shape_i, size_i, i = 1,\ldots, p$, expresses the shape of the gripper in terms of its number p of fingers, the shape and dimensions of each finger. *Rectangular*-shaped fingers are considered; their size is given "width" and "height".

- $fingers\_location(\mathcal{G}, \mathcal{O}) = \{x_{ci}(\mathcal{O}), y_{ci}(\mathcal{O}), rz_i(\mathcal{O})\}, i = 1,\ldots, p$,indicates the relative location of each finger with respect to the object's *centre of mass* and *minimum inertia axis* (MIA). At training time, this description is created for the object's model, and its updating will be performed at run time by the vision system for any recognized instance of the prototype.

- $fingers\_viewing(\mathcal{G}, pose\_context_i, i = 1,\ldots p)$ indicating the way how "invisible" fingers are to be treated; fingers are "invisible" if they are outside the field of view.

- $grip = 1,\ldots, k$ are the k gripper-object $\mathcal{G}s\_m(\mathcal{G}, \mathcal{O})$ distinct grasping models a priori trained, as possible alternatives to face at run time foreground context situations.

A *collision-free grasping transformation* $\mathcal{CF}(\mathcal{G}s\_m_i, \mathcal{O})$ will be selected at run time from one of the k *grip* parameters, after checking that all pixels belonging to $\mathcal{FGP}\_m_i$ (the projection of the gripper's fingerprints onto the image plane x_{vis}, y_{vis}, in the $\mathcal{O}$-grasping location) cover only background-coloured pixels. To provide a secure, collision-free access to objects, the following robot-vision sequence must be executed:

1. *Training* k sets of parameters of the multiple fingerprints model $\mathcal{MFGP}\_m(\mathcal{G}, \mathcal{O})$ for $\mathcal{G}$ and object class $\mathcal{O}$, relative to the k learned grasping styles $\mathcal{G}s\_m_i(\mathcal{G}, \mathcal{O}), i = 1,\ldots, k$.
2. *Installing* the multiple fingerprint model $\mathcal{MFGP}\_m(\mathcal{G}, \mathcal{O})$ defining the shape, position and interpretation (viewing) of the robot gripper for clear-grip tests, by including the model parameters in a data base available at run time. This must be done at the start of application programs prior to any image acquisition and object locating.
3. *Automatically performing the clear-grip test* whenever a prototype is recognized and located at run time, and grips $\mathcal{FGP}\_m_i, i = 1,\ldots.k$ have been a priori defined for it.
4. On line *call of the grasping parameters* trained in the $\mathcal{G}s\_m_i(\mathcal{G}, \mathcal{O})$ model, which corresponds to the first grip $\mathcal{FGP}\_m_i$ found to be clear.

The first step in this robot-vision sequence prepares off line the data allowing to position at run time two Windows Region of Interest (WROI) around the current object, invariant to its visually computed location, corresponding to the two gripper fingerprints. This data refers to the size, position and orientation of the gripper's fingerprints, and is based on:

- the *number and dimensions of the gripper's fingers*: 2-parallel fingered grippers were considered, each one having a rectangular shape of dimensions wd_g, ht_g;
- the *grasping location of the fingers relative to the class model* of objects of interest.

This last information is obtained by *learning* any grasping transformation for a class of objects (e.g. "LA"), and is described by help of Fig. 9. The following frames and relative transformations are considered:

- Frames: (x_0, y_0): in the robot's base (world); (x_{vis}, y_{vis}): attached to the image plane; (x_g, y_g): attached to the gripper in its end-point T; (x_{loc}, y_{loc}): default object-attached frame, $x_{loc} \equiv \text{MIA}$ (the part's minimum inertia axis); (x_{obj}, y_{obj}): rotated object-attached frame, with $x_{obj} \equiv \text{dir(C,G)}$, $\text{C}(x_c, y_c)$ being the object's centre of mass and $\text{G} = \text{proj}_{(x_{vis}, y_{vis})}\text{T}$;
- Relative transformations: to.cam[cam]: describes, for the given camera, the location of the vision frame with respect to the robot's base frame; vis.loc: describes the location of the object-attached frame with respect to the vision frame; vis.obj: describes the location of the object-attached frame with respect to the vision frame; pt.rob: describes the location of the gripper frame with respect to the robot frame; pt.vis: describes the location of the gripper frame with respect to the vision frame.

As a result of this learning stage, which uses vision and the robot's joint encoders as measuring devices, a grasping model $\mathcal{GP}\_m(\mathcal{G}, \text{"LA"}) = \{d.cg, alpha, z\_off, rz\_off\}$ is derived, relative to the object's centre of mass C and minimum inertia axis MIA (C and MIA are also available at runtime):

$$d.cg = \text{dist(C,G)},\ alpha = \angle(\text{MIA}, \text{dir(C,G)}), z\_off = \text{dist(T,G)}, rz\_off = \angle(x_g, \text{dir(C,G)})$$

A clear grip test is executed at run time to check the collision-free grasping of a recognized and located object, by projecting the gripper's fingerprints onto the image plane, (x_{vis}, y_{vis}), and verifying whether they "cover" only *background pixels*, which means that no other object exists close to the area where the gripper's fingers will be positioned by the current robot motion command. A negative result of this test will not authorize the grasping of the object.
For the test purpose, two WROIs are placed in the image plane, exactly over the areas occupied by the projections of the gripper's fingerprints in the image plane for the desired, object-relative grasping location computed from $\mathcal{GP}\_m(\mathcal{G}, \text{"LA"})$; the position (C) and orientation (MIA) of the recognized object must be available. From the invariant, part-related data:

$alpha, rz.off, wd_{\mathrm{LA}}, wd_g, ht_g, d.cg$, there will be first computed at run time the current coordinates $x_{\mathrm{G}}, y_{\mathrm{G}}$ of the point G, and the current orientation angle $angle.grasp$ of the gripper slide axis relative to the vision frame.

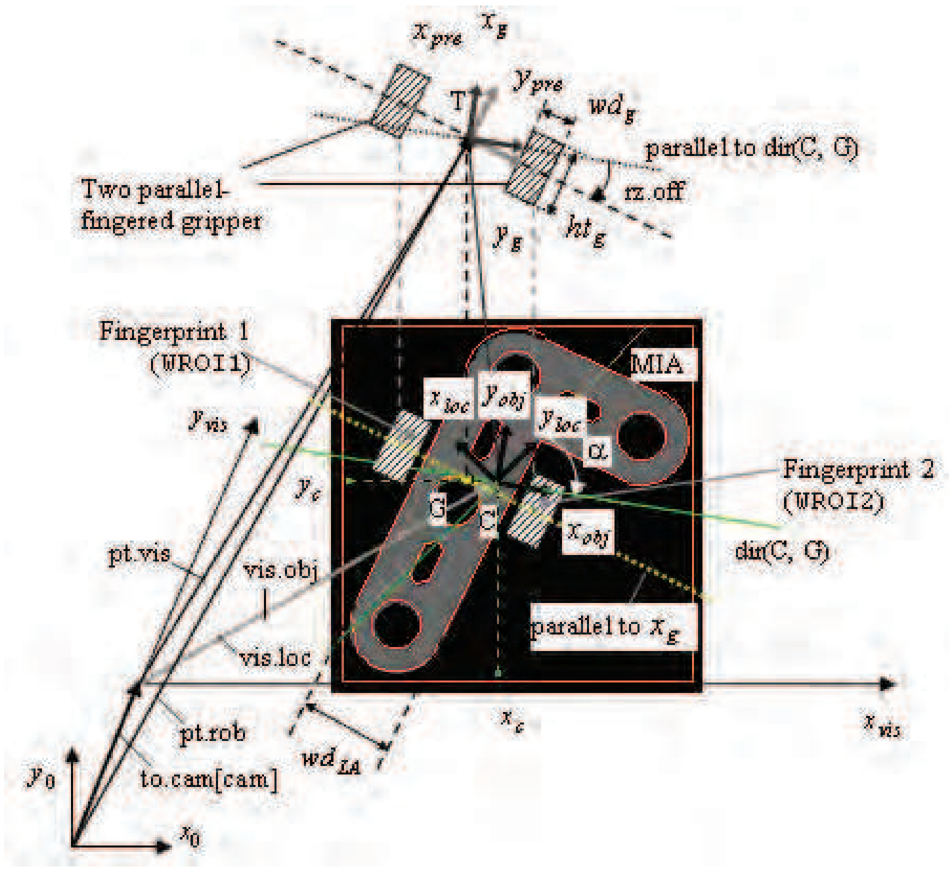


Figure 9. Frames and relative transformations used to teach the *GP_m(G,* "LA"*)* parameters

The part's orientation $angle.aim = \angle(\mathrm{MIA}, x_{vis})$ returned by vision is added to the learned $alpha$.

$$beta = \angle(\mathrm{dir(C,G)}, x_{vis}) = angle.aim + alpha \tag{5}$$

Once the part located, the coordinates $x_{\mathrm{C}}, y_{\mathrm{C}}$ of its gravity centre C are available from vision. Using them and *beta*, the coordinates $x_{\mathrm{G}}, y_{\mathrm{G}}$ of the G are computed as follows:

$$x_{\mathrm{G}} = x_{\mathrm{C}} - d.cg \cdot \cos(beta),\ y_{\mathrm{G}} = y_{\mathrm{C}} - d.cg \cdot \sin(beta) \tag{6}$$

Now, the value of $angle.grasp = \angle(x_g, x_{vis})$, for the object's current orientation and accounting for $rz.off$ from the desired, learned grasping model, is obtained from $angle.grasp = beta + rz.off$.

Two image areas, corresponding to the projections of the two fingerprints on the image plane, are next specified using two WROI operations. Using the geometry data from Fig. 9, and denoting by dg the offset between the end-tip point projection G, and the fingerprints centres CWi, $\forall i = 1,2$, $dg = wd_{\mathrm{LA}} / 2 + wd_g / 2$, the <u>positions</u> of the rectangle image areas "covered" by the fingerprints projected on the image plane in the desired part-relative grasping location are computed at run time according to (7). Their common orientation in the image plane is given by $angle.grasp$.

$$x_{\mathrm{cw1}} = x_{\mathrm{G}} - dg \cdot \cos(angle.grasp);\, x_{\mathrm{cw2}} = x_{\mathrm{G}} + dg \cdot \cos(angle.grasp)$$

$$y_{\mathrm{cw1}} = y_{\mathrm{G}} - dg \cdot \sin(angle.grasp);\; y_{\mathrm{cw2}} = y_{\mathrm{G}} + dg \cdot \sin(angle.grasp) \tag{7}$$

The type of image statistics is returned as the total number of non-zero (background) pixels found in each one of the two windows, superposed onto the areas covered by the fingerprints projections in the image plane, around the object. The clear grip test checks these values returned by the two WROI-generating operations, corresponding to the number of background pixels not occupied by other objects close to the current one (counted exactly in the gripper's fingerprint projection areas), against the total number of pixels corresponding to the surfaces of the rectangle fingerprints. If the difference between the compared values is less than an imposed error err for both fingerprints – windows, the grasping is authorized:

```
If $|ar1[4] - ar.fngprt| \le err$ AND $|ar2[4] - ar.fngprt| \le err$ ,
    clear grip of  object is authorized; proceed object tracking by continuously
    altering its target location on the vision belt, until robot motion is completed.
Else
    another objects is too close to the current one, grasping is not authorized.
```

Here, $ar.fngprt = wd_g ht_g / [(\text{pix.to.mm})^2 \, \text{XY\_scale}]$ is the fingerprint's area [raw pixels], using the camera-robot calibration data: pix.to.mm (no. of image pixels/1 mm), and XY_scale (x / y ratio of each pixel).

5. Conclusion

The robot motion control algorithms with guidance vision for tracking and grasping objects moving on conveyor belts, modelled with belt variables and 1-d.o.f. robotic device, have been tested on a robot-vision system composed from a Cobra 600TT manipulator, a C40 robot controller equipped with EVI vision processor from Adept Technology, a parallel two-fingered RIP6.2 gripper from CCMOP, a "large-format" stationary camera (1024x1024 pixels) down looking at the conveyor belt, and a GEL-209 magnetic encoder with 1024 pulses per revolution from Leonard Bauer. The encoder's output is fed to one of the EJI cards of the robot controller, the belt conveyor being "seen" as an external device.

Image acquisition used strobe light in *synchronous* mode to avoid the acquisition of blurred images for objects moving on the conveyor belt. The strobe light is triggered each time an image acquisition and processing operation is executed at runtime. Image acquisitions are synchronised with external events of the type: "*a part has completely entered the belt window*"; because these events generate on-off photocell signals, they trigger the *fast digital-interrupt line* of the robot controller to which the photocell is physically connected. Hence, the VPICTURE operations always wait on interrupt signals, which significantly improve the response time at external events. Because a fast line was used, the most unfavourable delay between the triggering of this line and the request for image acquisition is of only 0.2 milliseconds.

The effects of this *most unfavourable 0.2 milliseconds time delay* upon the integrity of object images have been analysed and tested for two modes of strobe light triggering:

- *Asynchronous triggering* with respect to the read cycle of the video camera, i.e. as soon as an image acquisition request appears. For a 51.2 cm width of the image field, and a line resolution of 512 pixels, the pixel width is of 1 mm. For a 2.5 m/sec high-speed motion of objects on the conveyor belt the most unfavourable delay of 0.2 milliseconds corresponds to a displacement of only one pixel (and hence one object-pixel might disappear during the *dist* travel above defined), as:

$$(0.0002 \text{ sec}) * (2500 \text{ mm/sec}) \,/\, (1 \text{ mm/pixel}) = 0.5 \text{ pixels.}$$

- *Synchronous triggering* with respect to the read cycle of the camera, inducing a variable time delay between the image acquisition request and the strobe light triggering. The most unfavourable delay was in this case 16.7 milliseconds, which may cause, for the same image field and belt speed a potential disappearance of 41.75 pixels from the camera's field of view (downstream the *dwnstr_lim* limit of the belt window).

Consequently, the bigger are the dimensions of the parts travelling on the conveyor belt, the higher is the risk of disappearance of pixels situated in downstream areas. Fig. 10 shows a statistics about the sum of:

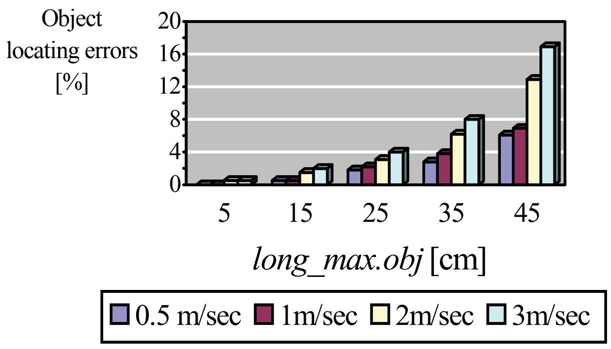
- *visual locating errors*: errors in object locating relative to the image frame (x_{vis}, y_{vis}); consequently, the request for motion planning will then not be issued;
- *motion planning errors*: errors in the robot's destinations evaluated during motion planning as being downstream *downstr_lim*, and hence not authorised,

function of the *object's dimension* (length *long_max.obj* along the minimal inertia axis) and of the *belt speed* (four high speed values have been considered: 0.5 m/sec, 1 m/sec, 2 m/sec and 3 m/sec).

As can be observed, at the very high motion speed of 3 m/sec, for parts longer than 35 cm there was registered a percentage of more than 16% of unsuccessful object locating and of more than 7% of missed planning of robot destinations (which are outside the CBW) for visually located parts, from a total number of 250 experiments.

The clear grip check method presented above was implemented in the V+ programming environment with AVI vision extension, and tested on the same robot vision platform containing an Adept Cobra 600TT SCARA-type manipulator, a 3-belt flexible feeding system Adept FlexFeeder 250 and a stationary, down looking matrix camera Panasonic GP MF650 inspecting the vision belt. The vision belt on which parts were travelling and presented to the camera was positioned for a convenient robot access within a window of 460 mm.

Experiments for collision-free part access on randomly populated conveyor belt have been carried out at several speed values of the transportation belt, in the range from 5 to 180 mm/sec. Table 1 shows the correspondence between the belt speeds and the maximum time intervals from the visual detection of a part and its collision-free grasping upon checking [#] sets of pre taught grasping models $\mathcal{G}s\_m_i(\mathcal{G}, O), i = 1,\ldots,\#$.



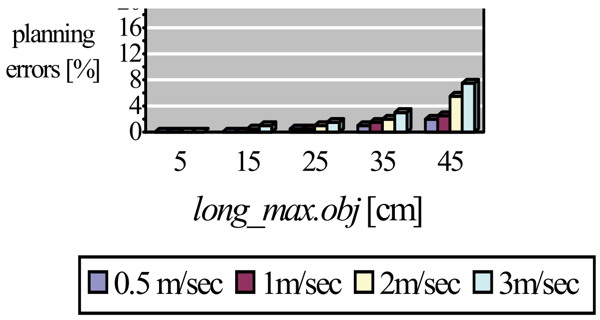


Figure 10. Error statistics for visual object locating and robot motion planning

Belt speed [mm/sec]	5	10	30	50	100	180
Grasping time (max) [sec]	1.4	1.6	1.9	2.0	2.3	2.5
Clear grips checked[#]	4	4	4	4	2	1

Table 1. Corespondance between belt speed and collision-free part grasping time

6. References

Adept Technology Inc. (2001). *AdeptVision User's Guide Version 14.0*, Technical Publications, Part Number 00964-03300, Rev. B, San Jose, CA

Borangiu, Th. & Dupas, M. (2001). *Robot – Vision. Mise en œuvre en V+*, Romanian Academy Press & AGIR Press, Bucharest

Borangiu, Th. (2002). Visual Conveyor Tracking for "Pick-On-The-Fly" Robot Motion Control, *Proc. of the IEEE Conf. Advanced Motion Control AMC'02*, pp. 317-322, Maribor

Borangiu, Th. (2004). *Intelligent Image Processing in Robotics and Manufacturing*, Romanian Academy Press, ISBN 973-27-1103-5, Bucarest

Borangiu, Th. & Kopacek, P. (2004). *Proceedings Volume from the IFAC Workshop Intelligent Assembly and Disassembly - IAD'03 Bucharest, October 9-11, 2003,* Elsevier Science, Pergamon Press, Oxford, UK

Borangiu, Th. (2005). Guidance Vision for Robots and Part Inspection, *Proceedings volume of the 14th Int. Conf. Robotics in Alpe-Adria-Danube Region RAAD'05*, pp. 27-54, ISBN 973-718-241-3, May 2005, Bucharest

Borangiu, Th.; Manu, M.; Anton, F.-D.; Tunaru, S. & Dogar, A. (2006). High-speed Robot Motion Control under Visual Guidance, *12th International Power Electronics and Motion Control Conference - EPE-PEMC 2006*, August 2006, Portoroz, SLO.

Espiau, B.; Chaumette, F. & Rives, P. (1992). A new approach to visual servoing in robotics, *IEEE Trans. Robot. Automat., vol. 8*, pp. 313-326

Lindenbaum, M. (1997). An Integrated Model for Evaluating the Amount of Data Required for Reliable Recognition, *IEEE Trans. on Pattern Analysis & Machine Intell.*

Hutchinson, S. A.; Hager, G.D. & Corke, P. (1996). A Tutorial on Visual Servo Control, *IEEE Trans. on Robotics and Automation*, vol. 12, pp. 1245-1266, October 1996

Schilling, R.J. (1990). *Fundamentals of Robotics. Analysis and Control*, Prentice-Hall, Englewood Cliffs, N.J.

Zhuang, X.; Wang, T. & Zhang, P. (1992). A Highly Robust Estimator through Partially Likelihood Function Modelling and Its Application in Computer Vision, *IEEE Trans. on Pattern Analysis and Machine Intelligence*

West, P. (2001). High Speed, Real-Time Machine Vision, *CyberOptics – Imagenation*, pp. 1-38 Portland, Oregon