OFF-LINE VERIFICATION OF AN INTELLIGENT CONTROL

V.S. ALAGAR
Department of Computer Science
Concordia University
Montreal H3G 1M8, Quebec, Canada

AND

K. PERIYASAMY
Department of Computer Science
University of Manitoba
Winnipeg R3T 2N2, Manitoba, Canada

1. Introduction

A computerized system is said to be safety critical if it never fails to perform as desired. Any departure from the required service at the required time constitutes *failure*. Failures are caused due to *errors*, which include mistakes in specification, design, environmental anomalies, and component failures. Not all errors produce immediate failures - errors may be propagated through several system states without causing a failure. Consequently, an absence of system failure during a certain span of system execution should not be interpreted as absence of errors in the system. In case of embedded software systems, an error may persist through several subsystem states and system cycles and may become noticeable only when its effect is seen in system failure. The consequences of such errors are generally hazardous and are difficult to correct.

There are many safety-critical applications in areas such as nuclear-power generation, transportation (aircraft, computerized train control signalling), autonomous process control systems (robotic-based explorations) and defence. For example, a control system for a pump feeding radioactive materials to a nuclear power generator is safety-critical, because failure in such a system could lead to serious accidents.

The general engineering approach to development of safety-critical systems is to identify potential hazards during requirements and design review and eliminate those leading to unacceptable risks. Several cycles of re-specification of requirements and redesign must be done to ensure that all requirements leading to risks are eliminated and safety features are incorporated in the design. The initial goal is to establish some quality criteria to be met by component parts (so that they are safe) and the environment (so that no stimulus can be an anomaly). The next goal is to demonstrate that the software specification is error free. It is this latter aspect that we discuss in this chapter.

In this chapter, we have chosen the control system for an automated mechanical parts assembly system for illustration. This control system is safety-critical because failures in this system should not cause damages to parts and machinery. Due to its constant interaction with external environment and time constrained requirements, the problem is also an instance of a real-time reactive system. We have given the design specification for the control system which consists of two major subsystems. Each subsystem consists of several interacting components. We show that the design specification achieves the level of assurance necessary to establish the *safety* and *liveness* properties.

An intelligent control system differs from a traditional (non-intelligent) control system in at least two ways: (i) An intelligent control has decision making capabilities. It makes decisions based on time dependent facts available to it and the constraints imposed on it by its environment; and (ii) During its operation, it derives new facts and rules that are useful for prediction and for future applications. To function intelligently an intelligent control must possess an inference mechanism to derive new facts and rules. Hence, the designer of an intelligent control is obliged to verify that the control derives only valid facts and rules pertinent to the application. That is, the operations of an intelligent control must be proved to be correct. This requires a proof of correspondence between the control system's specification and the software itself (say, written in some high-level language). Because we do not address programming issues in this chapter, we provide a rigorous justification to establish that the (design) specification of the assembly control implements the requirements specification.

The behavior of control systems can be verified either online during its actual usage in an application or off-line using its specification. In an online verification, it is difficult to verify all properties in a bounded time and hence verification is quite expensive. The major advantage of off-line verification is that proofs of varying degrees can be conducted on each level of specification, and the final code itself, to declare whether or not safety and liveness properties are captured in each level of specification. The software is released for use only after off-line verification is completed. A system can be verified (even if the process is only informal) only if its behavior is formally expressed using some mathematical notations. Verification based on informal specifications will be much less effective than informal verification based on formal specification.

Formal methods are increasingly used [9] in the design and development of safety-critical systems. In fact, the Department of Defence at the United Kingdom requires that any software procured by the Defence organization in the U.K. must be accompanied by its formal specification. There are two aspects to formal methods: the *formal* aspect, and the *method* aspect. The first aspect concerns the language elements, and the basic concepts of mathematics and logic on which the formalism is founded. The language constructs are used to formally document requirements of the chosen application and specify its design. The second aspect provides a methodology for writing the specification, lends support for deriving successive refinements and establish a proof of correspondence between one level of specification to its superior level of specification. In such a formal framework, everything that can be assumed about every specified entity must follow from the assertions that make up the specification and by the inference system of the language. Our knowledge and intuition about the problem domain are used in developing the specifications so that the specified system acts intelligently thereafter. The benefits of using formal methods are that they reduce ambiguity and imprecision in a model, and provide specifications that can be rigorously analyzed for flaws and tested for desirable properties. Tools based on the methodology will facilitate formal verification process. In the absence of tools, an informal verification based on the formalism should be acceptable.

Recognizing the great benefits of applying formal methods very early in the life cycle we smoothly harness the formalism with object-oriented (OO) methodology. An object-oriented approach to software design differs from the traditional, functional approach in many ways: (i) OO design emphasizes on objects and their local behaviors while the functional approach looks at the functionalities or tasks to be performed in the entire system; (ii) Maintenance, and enhancement are easier to perform in an OO design; and (iii) Techniques such as inheritance and aggregation in OO designs promote software reuse. We have chosen the formal method Object-Z [6] due to two reasons: (i) An implementation from Object-Z specification can be derived using the work [7]; and (ii) Object-Z notation meshes well with object-oriented modeling derived from an analysis of the problem.

Since the controller for the assembly cell is a reactive system, we have to specify both its functional behavior as well as its real-time constraints. The Object-Z notation does not support real-time specifications. Therefore, we have extended the Object-Z notation with additional syntax in order to express timing constraints. In this process, we have introduced *filter* *specifications*. A filter specification describes the mapping between timing variables and functional variables. It is separated from the functional specification in Object-Z, but uses the same notation and structure as that of an Object-Z specification. Our approach is significant due to the following reasons: (i) System functionality is separated from its real-time constraint so that changes can be managed independently. Using the formal semantics [2] these descriptions can be combined to produce the timed structure of the system; (ii) the notation for the filter specification is the same as that of Object-Z; thus understanding the specifications and proving desired properties from the specifications become easier; and (iii) the filter specification also enjoys the benefits of object-orientation such as inheritance and encapsulation.

The rest of the chapter is organized as follows: We give an overview of object-orientation in Section 2. In Section 3, we give the notations for Object-Z and the filter specification. The semantics of the notations is illustrated with simple example in this section. Section 4 describes the assumptions and requirements of the case study (assembly cell) along with the timing requirements. These requirements include safety and liveness properties. Next, we describe a complete specification of the subsystems in the case study in Section 5. We informally prove the safety and liveness properties using the specifications in Section 6. The chapter concludes in Section 7 with comments on future work.

2. Object-Oriented Modelling

In this section, we first give a brief introduction to the concepts of object-orientation. We restrict ourselves to a subset of OO concepts which are necessary to understand the specifications given in a later section.

2.1. OBJECT-ORIENTATION

The object-oriented approach insists on describing each object in the problem domain in its entirety. This means that the structure and behavior of each object are described in an encapsulated manner. The structure is defined by a set of data variables (called *attributes*) and the behavior is described by a set of procedures and/or functions (called *methods*). Different notations and terminologies are in use for describing the structure and behavior of an object [5, 11, 13].

Following are the definitions of some concepts in object-orientation, which we use:

Object An object represents a real-world entity in the problem domain.
For example, the object-oriented model of a robot will consist of the

objects *arm, finger, joint* and *base*. In addition, objects may also be created to represent system entities during modeling. For instance, the object *coordinate frame* may be created during the modeling of a robot. An object has a structure and behavior which are called *attributes* and *methods* respectively. They are collectively known as *features* of the object. The enclosure of its features within an object is called *encapsulation*.

Attribute A data member describing a structural component of an object is an attribute. For example, *length* is an attribute of the object *arm*. An object can only have a finite number of attributes.

Behavior The behavior of an object can be observed by executing its methods. A method describes an operation that could be performed on the structure of an object. For example, *change_position* is a member of the object *arm*. Methods may also represent the communication between objects. For instance, *arm* may have a method *convert_position* which converts its position, given in its local coordinate frame, to a position with respect to the global coordinate frame. In its strict sense, the structure of an object can only be accessed and modified by a method of the same object. However, several object-oriented programming languages such as C++ leave it to the programmer to make judicial choices on this issue.

Class A collection of objects having identical structure and behavior are grouped into a *class*. A class also serves as a type definition for the collection of objects it represents. Consequently, every object belongs to exactly one class definition. It is interesting to note that programmers create only class definitions and objects are created by the system at run-time. In a typical object-oriented programming language, an object *leftarm* is represented as a variable of type *arm*, the latter being a class. The object *leftarm* is said to be an instance of *arm*. Objects of the same class will have distinct identifications and different values for the attributes but share the same code for the methods in their class.

Exported Features Due to encapsulation, a class has the right to expose part or whole of its features and hide the rest. These exposed features are said to be exported by the class, and are visible to other classes in the system. Accordingly, objects of other classes can communicate or access (defined next) only these exported features. The hidden features are generally used to implement the exported features and are local to the defining class. Consequently, changes in hidden features will not affect the structure and behavior of the class, known to the system environment. As an example, the class *arm* may export the methods *change_position* and *convert_position* and hide its structural components *position* and *length*.

Message Passing Objects communicate with each other by passing messages. Typically, message-passing is accomplished by a client-server relationship. Accordingly, a *controller* object wishing to call the method *change_position* in the object *leftarm* will send a message to *leftarm*. It is the responsibility of *leftarm* to provide the required service or reject the request for some reason. The *controller* object can request *leftarm* for *change_position* only if *change_position* is an exported feature of *leftarm*.

Inheritance One of the major distinguishing features of object-oriented approach compared to other traditional approaches is that a class c_{sub} can be defined as a specialization of another class c_{sup} by inheriting the structure and behavior of c_{sup}. The class c_{sup} in this case is called the *superclass* and c_{sub} is called the *subclass*. Besides inheriting, the subclass can also rename or redefine the methods inherited from the superclass. For example, the class *multi_fingered_arm* can be defined as a subclass of *arm*. The former will acquire all features of *arm* and define additional structure and behavior corresponding to the fingers.

Polymorphism Another distinguishing feature of the object-oriented approach is the ability to choose an implementation of a method from among its different implementations. This aspect is called *polymorphism*. Accordingly, a method m in a class can be multiply defined within the same class or redefined when inherited into a subclass. The conditions for such multiple definitions depend on the choice of the programming language. For example, the method *convert_position* can be multiply defined within the class *arm* to return the result in different format, or redefined by the subclass *multi_fingered_arm* in order to include the positions of the fingers.

3. Specification Notation

This section gives an introduction to the formalism that we use to formally specify the model created in Section 3.

3.1. OBJECT-Z NOTATION

A specification in Object-Z consists of a set of global definitions and a set of class definitions. The set of global definitions are shared by all class definitions. A class definition is encapsulated; i.e., the paragraphs within a class definition are local to the class itself. Global definitions and the paragraphs within each class definition are given using Z notation. Thus, Object-Z is an extension of the Z notation. The structure of a class definition is given in Figure 1.

ClassName

$\upharpoonright$(*visibility list*)
list of inherited class names
type definitions
constant definitions
state space schema
initial state space schema
operation schemas

Figure 1. Structure of a class definition in Object-Z

The *visibility list* of a class C lists all the features that are exported from C. If omitted from the definition, all features of C are visible. Typically, this list includes attributes and operations of C. The *list of inherited class names* of C indicates all the superclasses of C. While inheriting a superclass C_{sup} into C, the *state space schema*, *initial state space schema* and *operation schemas* of C_{sup} are conjoined with the respective components in C according to schema conjunction in Z. Redefinition of an operation can only provide additional constraints and/or introduce new parameters. The *visibility list* of C_{sup} is not inherited into C. Hence C can export an inherited feature from C_{sup}, although C_{sup} does not export that feature. *Type definitions* and *constant definitions* are similar to those in Z, but are local to the class. The *state space schema* is unnamed; the declaration part of the *state space schema* corresponds to attributes of the class. The *initial state space schema* describes the set of all initial states of objects instantiated from the class.

Operation schemas are defined as in Z, with one exception. While an operation in Z includes the state spaces which are affected by the operation using the Δ or Ξ notation, the state space is implicit for an operation in Object-Z; the state space of the class. The notation $\Delta(x, y)$ in an operation *Op* in Object-Z asserts that the attributes x and y are affected by *Op*; other attributes of the class are unaffected by *Op*. This is a major semantic difference between Z and Object-Z.

Operations in Object-Z can also be defined by composing several other operations. There are four composition operators in Object-Z. These are: *conjunction* ($\wedge$), *parallel* ($\|$), *sequential composition* ($\fatsemi$) and *choice* ($[]$). The expression $Op_1 \wedge Op_2$ denotes an operation schema in which the operations Op_1 and Op_2 are independent and will be invoked concurrently; common declarations in both operations are equated. The expression $Op_1 \parallel Op_2$ denotes hand-shaking communication between the operations Op_1 and Op_2.

So, input variables of one operation are equated to the output variables of the other as long as they have the same base name (e.g., $x?$ and $x!$). The expression $Op_1 \mathbin{\S} Op_2$ denotes sequential composition in which Op_2 follows Op_1. Accordingly, Op_2 proceeds only after Op_1 terminates; the notion of intermediate state is evident. The expression $Op_1 [] Op_2$ denotes an operation in which either Op_1 or Op_2, but not both, will be enabled depending on the pre-condition of whichever operation is satisfied. If both the pre-conditions are satisfied, one of the operations is non-deterministically chosen.

Object-Z notation is case-sensitive. Hence, the identifiers *Recognize* and *RECOGNIZE* are different; the former may denote an operation while the latter may denote a value of an enumerated constant.

3.2. FILTER SPECIFICATION

A filter schema defines an association between variables (data), processes and real-time. Additional timing constraints as mandated in the problem domain that affect the variables and processes in a filter may also be defined in a filter schema. A filter specification is a collection of filters.

The structure of a filter specification is the same as that of a class definition except that it has only a set of filters. Each filter specification is associated with exactly one class definition. For ease of identification, we give the name of filter specification of a class C as C**_filter** (the word **filter** is a keyword). Each filter schema corresponds to an operation in the class. The name of the filter is the same as the name of the operation in C to which it corresponds. This syntax is shown in Figure 2.

A filter schema Op_1 of C**_filter** corresponding to the operation Op_1 of C lists a set of functional variables affected by the filter and four timing parameters: t, referring to the current time at which the filter and its operation are invoked; δt, referring to the anticipated completion time of the operation; ϵ_{1x}, denoting the offset from the current invocation of the filter (i.e., $t + \epsilon_{1x}$) at which an input x, if any, is received by the operation; and ϵ_{2y}, denoting the offset from the current invocation after which the operation will output a value y, if any. There are two implicit constraints on the values of ϵ_{1x} and ϵ_{2y}; they are:

$$0 \leq \epsilon_{1x} < \delta t \wedge 0 < \epsilon_{2y} \leq \delta t$$

These two constraints assert that an input value can be received at the time of invocation or any time later but before the operation is completed while a value is output strictly sometime after the operation is invoked.

C

...

Op_1

...

...

Op_n

...

C**_filter**

Op_1

...

...

Op_n

...

Figure 2. A Class and its Filter Specification

The variable t is of type *RealTime* and it denotes an instance of a global clock. The other three variables δt, ϵ_{1x} and ϵ_{2y} take metric values and are of type *Time*. The type *RealTime* and *Time* are coercive with respect to addition and subtraction and the result of coercion is of type *RealTime*; for example, the expression $t + \delta t$ will be of type *RealTime*. These timing parameters augment data variables in the specification, thus realizing data sensitive real-time constraints. A filter constrains its associated operation, thus realizing process sensitive real-time constraints.

The predicate part of a filter consists of two predicates: The first predicate introduces a map, identified by the keyword 'mapping' which provides the mapping between the functional variables and the timing variables in the declaration of the filter, and the global timing constants of the filter specification. These global timing constants are derived from problem domain, will be specified globally outside the filter specifications and are accessible by all the filter specifications. The second predicate in a filter describes the relationship among the timing variables in this filter. This predicate is optional.

Remarks: Decoupling the functional (process) descriptions from the timing constraints has several advantages to the design: (1) the specifications of functional components and their refinements remain reusable in contexts where time constraints are not relevant; (2) changes to the functionalities can be analyzed and respecified independent of timing constraints (which need not change); (3) time dependent properties can be better understood by inspecting the specifications in one tier; (4) both process-oriented real-time constraints and data sensitive real-time constraints are uniformly treated by filters;and (5) a filter specification also enjoys the benefits of OO methodology.

4. Case Study - A Robot-based Assembly Cell

This section presents a case study in robotics. The assembly cell consists of a robot with two arms, several conveyor belts, several sensors, a camera and an assembly pad. The task is to model a control system satisfying the requirements as stated below. From the formal specification of the model it should be possible to verify safety and liveness properties.

The assembly cell consists of a conveyor belt called *feeder*, a rotary belt, three channel belts called *ChanC*, *ChanD* and *ChanR*, a camera, a robot with two identical arms, and a rotary table, one end of which is used as the base, to perform assembly operations. There are five sensors used in the assembly cell. Parts are fed through the *feeder* line. When a part is under the camera, the type of the part is recognized soon after which the rotary belt receives this information. The rotary belt, in turn, aligns itself with one of the three channels and feeds the part to the end of the channel belt. *ChanC* and *ChanD* are reserved for the two types of parts that can be assembled. The robot arms pick up the parts and place them on the assembly pad. After performing an assembly, the rotary table removes the assembled product from the pad. Parts that do not flow into *ChanC* or *ChanD* are diverted to *ChanR*. A top view of the assembly cell is shown in Figure 3.

From the above description, it is clear that the case study belongs to *safety-critical systems*. A safety property is that parts should not collide and get damaged. A liveness property is that eventually all assemblies are made regardless of the order of arrival of parts at the robot pick-up point. The system also requires real-time constraints (see Section 4.3) to be embedded in the design in order to ensure safety and liveness properties. According to one of the constraints, the rotary belt has to divert a part to appropriate channel and then re-orient itself before the next part arrives from the feeder. By virtue of these requirements, the chosen problem is inherently complex and requires a thorough behavioral study before it is designed and implemented. Such a behavioral study requires a formal notation to

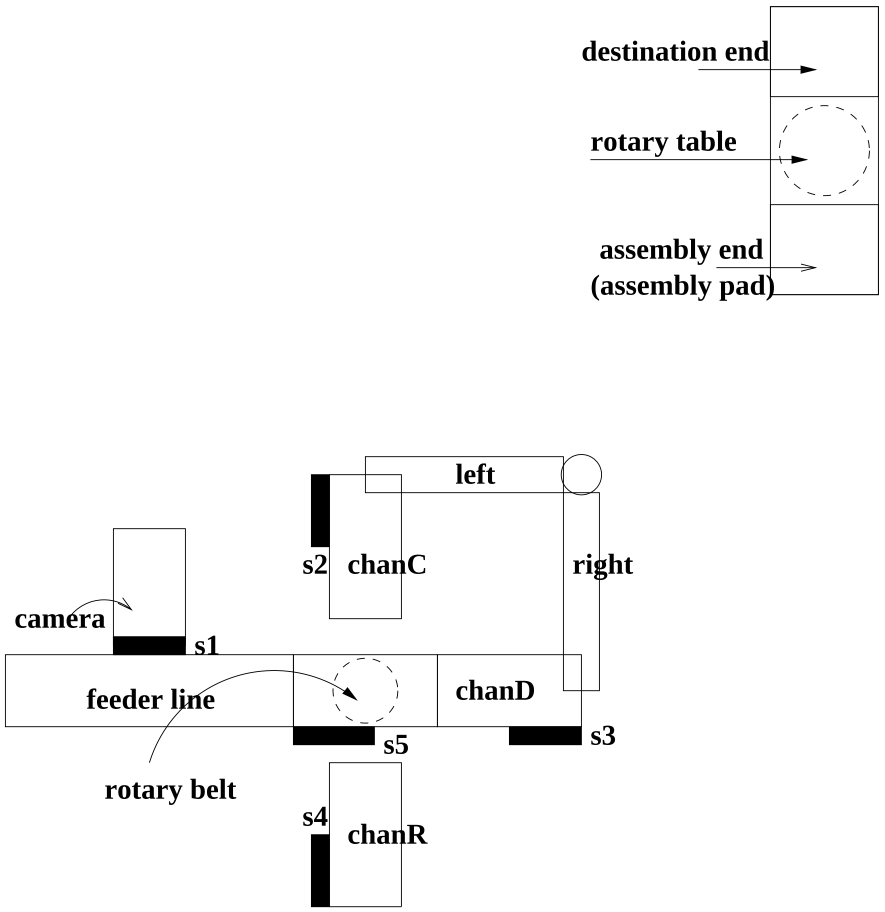


Figure 3. Top View of the Assembly Cell

express the desired properties and ensure that (i) the system is well understood, (ii) the safety and liveness properties are captured in the design and (iii) a modular and easy-to-enhance design can be derived from the specification. See [10] for a collection of 18 different contributions to the modeling, specifications, and verification approaches to a similar Production Cell Problem. It is a very challenging task to compare our approach with any of those approaches; our approach significantly differs from these approaches. The major differences are the choice of specification notation, the way timing constraints are decoupled in the design, and the logic that we use [2] towards formal verification.

4.1. INFORMAL DESCRIPTION OF THE ASSEMBLY PROCESS

For the sake of simplicity, we refer to the two types of parts that can be assembled as *Cup* and *Dish*, even though any such pair can be assembled. Parts are placed on the *feeder*, one at a time at the prescribed position. When a part comes in front of $s1$, camera starts recognizing the part. The vision control communicates with the rotary belt mechanism to inform the type of the part being recognized. If a cup is received, the rotary belt rotates 90° (anti-clockwise) to align itself with *ChanC*. Then both the rotary belt and the channel belt move to bring the cup to the tail end of $ChanC$. Sensor $s2$, recognizing the arrival of a cup at the tail end of ChanC, stops both the belts. The robot's left arm, if free, will pick up the cup. Soon after the cup reaches the end of *ChanC*, the rotary belt re-aligns itself with the feeder and is ready to receive the next part. If the part recognized by the camera is a dish, it is moved to *ChanD* with no rotation of the rotary belt. A dish reaching the end of *ChanD*, recognized by the sensor $s3$, will eventually be picked up by the right arm of the robot, if it is free.

Parts which are neither cups nor dishes are routed through *ChanR*; these parts will manually be removed at the end of *ChanR*. There are situations when a recognized cup or dish will also be returned through *ChanR*. This occurs when there is already a cup or dish waiting at the tail end of *ChanC* or *ChanD* and the next cup or dish arrives at the rotary belt. It is assumed that these returned cups and dishes will be re-fed within a (finite) prescribed time.

When the assembly pad is empty, the right arm of the robot, if it has a dish, will place the dish. The left arm, if it holds a cup, will place the cup only after a dish has already been placed on the assembly pad. After performing an assembly, the assembly table is rotated through 180° so that the assembled part is delivered to the other end. The free end of the rotary table will then become the assembly pad to perform the next assembly.

4.2. ASSUMPTIONS

The following assumptions are made in developing the specification. For the sake of clarity and understandability, the assumptions are classified and stated for each component of the assembly system.

Assumptions on Environment

- The assembly cell is a 2-D environment.
- All reference positions (for parts, robots etc.) are points in the environment. They are described only with respect to a global coordinate frame.

Assumptions on Parts

- There are only two types of parts that can be assembled; an assembly consists of one part from each type. We use the terms 'Cup' and 'Dish' to denote the two types.
- Parts are fed one at a time.
- Successive parts on the feeder will maintain a constant distance d between them.
- Cups and dishes can be fed in any order. The number of cups and dishes to be fed to the assembly cell will be known initially.
- The geometry of a part is ignored since it is not relevant for the current discussion. Hence a part is characterized by its position and its status (it is on a belt or it has been grasped by a robot arm or it has been placed on the assembly pad).

Assumptions on Camera

- A camera is used to detect the types of parts. It is assumed that the camera will correctly recognize a cup or a dish when placed in front of the camera regardless of the orientation of the part.
- Details of recognition process such as object recovery from images and pattern matching will be ignored. Instead, recognition process will be considered atomic.

Assumptions on Belts

- A belt is at rest until activated to move.
- All belts move uniformly (with no jerks) at a constant speed.
- The rotary belt and the three channel belts are all of same length d.
- The length of the feeder is $2 * d$. The camera is located at a distance d from the start of the belt.
- Parts reaching the end of a belt are assumed to smoothly transit to the other side of the belt, with no damage to the part. In addition, the orientation of a part is unaltered due to a belt's motion and when the part transits between belts.
- If two belts b_1 and b_2 touch each other, say the start of b_2 coincides with end of b_1, then the spatial coordinates of the two positions referring to the start of b_2 and end of b_1 are the same. Consequently, any part resting on the end of b_1 is considered to have moved into b_2. That is, the part is assumed to to be on b_2 and is no longer on b_1.

Assumptions on Robot

- Both arms of the robot have identical features.
- Details of arm structure such as number of links, type of finger and degrees of freedom are ignored; these details are not important for the discussion at this level of abstraction.

- The position of an arm refers to the position of the fingers in that arm. Consequently, when a part is picked up by an arm, the position of the arm coincides with the position of the part.
- Each arm has an initial position where it stays when there is no part to pick up or when the arm is waiting. A pick operation is atomic and is composed of the sequence of tasks: *move from the initial position to the grasping position, reorient the fingers for grasping, pick up a part and move back to the initial position.* Similarly, a place operation is also atomic and is composed of the sequence of tasks: *move from the initial position to the assembly pad, reorient the fingers for dropping, drop the part on the assembly pad and move back to the initial position.* We do not specify these individual tasks; instead, specify the two atomic operations - pick and place.

Assumptions on Assembly

- For simplicity of the specification, details of the actual assembly process such as gluing, screwing or pressing are ignored. A specification of an assembly in terms of surface contacts of parts is given in [12].

4.3. TIMING CONSTRAINTS FOR THE CASE STUDY

Several timing constraints apply to the tasks and the intermediate computations. We define these by the following real-valued constants:

s	-	speed of a belt.
d	-	length of the belts: rotary_belt, chanC, chanD and chanR.
t_r	-	time taken by the camera to recognize a part.
t_s	-	time during which the feeder belt is stopped.
t_d	-	time taken by a part to travel a distance d.
t_{pi}	-	time taken by a robot arm to pick up a part; this includes the time taken to move the arm from its initial position to the pick up point, orient for grasping, grasp the part and return to the initial position.
t_{pl}	-	time taken by a robot arm to place a part; includes the time taken to move the arm from its initial position to the destination, orient for placing, place the part and return to the initial position.
t_{ro}	-	time taken by the rotary belt to rotate through 90°.
t_{ao}	-	time taken by the rotary assembly table to rotate through 180°.
t_{as}	-	time taken to perform one assembly.
t_{fb}	-	minimum time taken to return a cup or a dish to the feeder line after it is returned through channel.

There are some restrictions on the values of the timing constants:

- The time t_d is derived from d and s; i.e., $t_d = d \ / \ s$.
- The feeder line should be stopped for the duration within which the camera recognizes a part and informs the rotary belt control; i.e.,
  $t_s = t_r$
- The rotary belt must be realigned with the feeder line before the next recognized part arrives at the rotary belt; i.e.,
  $t_s + t_d \geq 2 * t_d + 2 * t_{ro}$
  $\equiv t_s \geq t_d + 2 * t_{ro}$
- A returned part should be re-fed before the next part is returned.
  $t_{fb} < t_{ro} + 2 * t_d$

4.4. SAFETY AND LIVENESS PROPERTIES

The following are the safety and liveness properties required to be satisfied in the design:

Safety properties:

- No collision occurs between parts when they are on belts.
- No two parts of the same kind are placed on the assembly pad.

Liveness properties:

- If n cups and n dishes are placed on the feeder, then exactly n assemblies are made within a finite amount of time.

4.5. OBJECT-ORIENTED DESIGN OF THE CASE STUDY

The object-oriented design for the case study consists of classes - *Part, Belt, Sensor, Camera, Robot Arm, Assembly Pad* and *Rotary Table*, where each class models one component in the system. The class *Belt* is a generic description for all the three channel belts, the rotary belt and the feeder. For simplicity, we group the rotary belt and the three channel belts into another class named *RotaryBeltChannels*. The two arms of the robot are instances of the class *RobotArm*. Similarly, all the five sensors are instances of the class *Sensor*.

The system model is simplified when we group the components into two subsystems - *Parts Recognition Subsystem* and *Assembly Subsystem*. Figures 4 and 5 show the various components in the two subsystems and their communication. As seen from these figures, there are components duplicated in each subsystem which serve as the links between the subsystems.

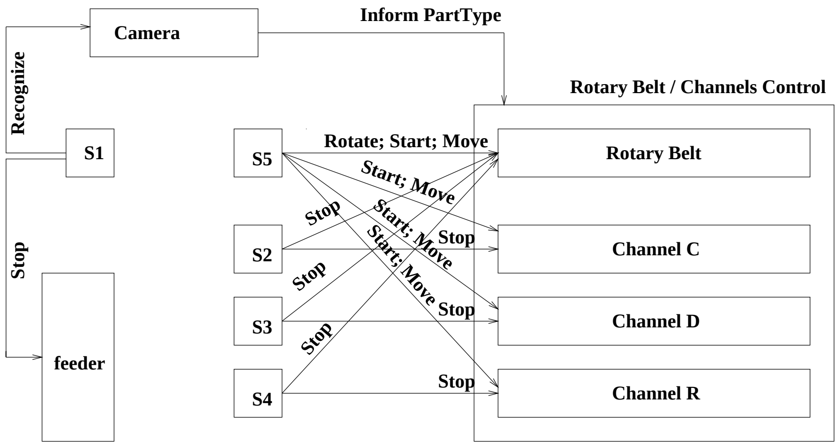


Figure 4. Architecture of the Parts Recognition Subsystem

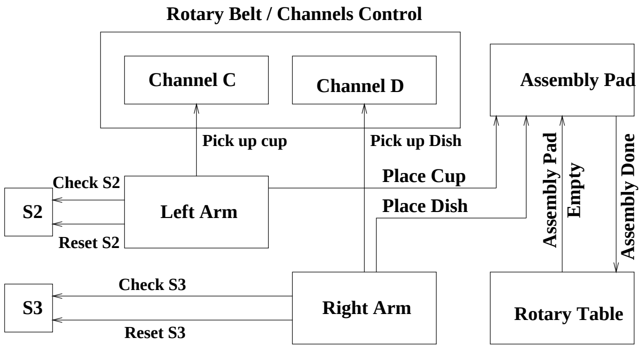


Figure 5. Architecture of the Assembly Subsystem

5. Formal Specification of the Case Study

In this section, we give the behavioral specification of two subsystems modeling the case study. Refer [1] for the specifications of individual classes and filters associated with them. At the end of this section, we illustrate the application of a filter specification to a class definition which generates the timed class specification. Following this method, the timed structure for the full system can be obtained.

5.1. OBJECT-Z SPECIFICATION

A formal behavioral specification of the whole system can be obtained by composing individual class definitions of basic components. Although we have given only the specifications for the subsystems in this chapter, for the sake of understanding, we first give the global types and variables definition and then informally describe the functionalities of the classes used in subsystem specifications. Type definitions and constants needed by all classes are shown in Figure 6.

$PartType ::= CUP \mid DISH \mid OTHER$
$PartStatus ::= ON\_BELT \mid GRASPED \mid ON\_ASSEMBLY\_PAD \mid$
$ASSEMBLED \mid IN\_COLLECTOR$
$CameraStatus ::= RECOGNIZE \mid IDLE$
$ArmStatus ::= HOLDING \mid EMPTY$
$BeltStatus ::= MOVING \mid STOPPED$
$RotaryBeltPosition ::= ZERO \mid NINETY \mid TWO\_SEVENTY$
$SensorStatus ::= ON \mid OFF$
$Direction ::= CLOCKWISE \mid ANTI\_CLOCKWISE$
$Assembly\_Pad\_Status ::= FREE \mid DISH\_IS\_THERE \mid PAIR\_IS\_THERE$
$Boolean ::= TRUE \mid FALSE$

$constant\_distance : \mathbb{R}$
$camera\_position : Point$
$collector\_position : Point$
$assembly\_pad\_position : Point$
$leftarm\_initial\_position : Point$
$rightarm\_initial\_position : Point$

Figure 6. Type Definitions and Constants

The status of a part indicates its position in the environment; the term *IN_COLLECTOR* denotes the status of a part when it is collected at the tail end of the *ChanR*. The type *RotaryBeltPosition* indicates the position of the rotary belt; it can be rotated into only three positions - ZERO (used to reset), NINETY (to feed *ChanC*) and TWO_SEVENTY (to feed *ChanR*). Other type definitions are self-explanatory. Positions in the system are specified using the type *Point*, which is assumed to be a basic type.

5.1.1. *Specifying Classes*

Due to space limitations, we give only a summary of the individual class definitions in Table 1. For details of these definitions, refer to [1].

5.1.2. *Specifying Subsystems*

Having defined the individual classes, we now give the specification for the two subsystems - *Parts Recognition Subsystem* and *Assembly Subsystem*, obtained by composing the classes specified in [1].

Class Name	Operation	Informal Description
Part	ChangePosition	to change the position of the part
	ChangeStatus	to change the status of the part
Camera	Recognize	to recognize a part
	Idle	to reset the camera
Belt	Start	to start the belt
	Stop	to stop the belt
	Move	to move the belt through a specified distance
	Place	to place a part at the beginning of the belt
RotaryBeltChannels	RotateNinety	to rotate the rotary belt and one of the channel belts through 90°
	Start_Belts_RBC	to start the rotary belt and *ChanC*
	Start_Belts_RBD	to start the rotary belt and *ChanD*
	Start_Belts_RBR	to start the rotary belt and *ChanR*
	Stop_Belts_RBC	to stop the rotary belt and *ChanC*
	Stop_Belts_RBD	to stop the rotary belt and *ChanD*
	Stop_Belts_RBR	to stop the rotary belt and *ChanR*
	Move_Belts_RBC	to move the rotary belt and *ChanC* through a specified distance
	Move_Belts_RBD	to move the rotary belt and *ChanD* through a specified distance
	Move_Belts_RBR	to move the rotary belt and *ChanR* through a specified distance
Sensor	ChangeStatus	to change the status of the sensor
RobotArm	Pick	to pick up a part from one of the channels
	Place	to place a part on the assembly pad
AssembledPart	ChangePosition	to change the position of an assembled part
AssemblyPad	PerformAssembly	to perform an assembly
RotaryTable	Rotate	to rotate the assembly pad in order to remove the assembled part

TABLE 1. Summary of Class Definitions

The specification for *Parts Recognition Subsystem* is shown in Figures 7 and 8. The specification asserts in a formal declarative manner the recognition of parts and the movement of parts to appropriate locations for the robot to pick up. There are four sections in the specification: the first section introduces variables modeling feeder belt, rotary belt channels, camera and sensor; the invariance of the subsystem is stated in the next section, followed by the initial configuration of the system; the last section specifies the operations.

$$
\begin{array}{l}
\textit{PartsRecognitionSubsystem} \\
\hline
\textit{feeder} : \textit{Belt} \\
\textit{rbc} : \textit{RotaryBeltChannels} \\
\textit{camera} : \textit{Camera} \\
s1, s2, s3, s4, s5 : \textit{Sensor} \\
\hline
\textit{distance}(\textit{feeder}.\textit{start\_of\_belt}, \textit{camera}.\textit{position}) = \\
\quad \textit{distance}(\textit{feeder}.\textit{end\_of\_belt}, \textit{camera}.\textit{position}) = \textit{constant\_distance} \\
\textit{feeder}.\,\mathsf{IsOnBelt}\,(\textit{camera}.\textit{position}) \\
(\forall\, i, j : 1 \mathinner{.\,.} \#(\textit{feeder}.\textit{objects}) \mid \textit{abs}(i - j) = 1 \bullet \\
\quad \textit{distance}((\textit{feeder}.\textit{objects}(i)).\textit{position}, (\textit{feeder}.\textit{objects}(j)).\textit{position}) \\
\qquad = \textit{constance\_distance}) \\
\{ \textit{obj} : \textit{Part} \mid \textit{obj} \in \operatorname{ran} \textit{feeder}.\textit{objects} \bullet \textit{obj}.\textit{id} \} \cap \\
\quad \{ \textit{obj} : \textit{Part} \mid \textit{obj} \in (\operatorname{ran} \textit{rbc}.\textit{rotary\_belt}.\textit{objects} \cup \\
\qquad \operatorname{ran} \textit{rbc}.\textit{chanC}.\textit{objects} \cup \operatorname{ran} \textit{rbc}.\textit{chanD}.\textit{objects} \cup \\
\qquad \operatorname{ran} \textit{rbc}.\textit{chanR}.\textit{objects}) \bullet \textit{obj}.\textit{id} \} = \varnothing \\
\textit{rbc}.\textit{rotary\_belt}.\textit{tablepos} = \textit{ZERO} \Rightarrow \\
\quad \textit{feeder}.\textit{end\_of\_belt} = \textit{rbc}.\textit{rotary\_belt}.\textit{start\_of\_belt} \\
s1.\textit{id} \neq s2.\textit{id} \neq s3.\textit{id} \neq s4.\textit{id} \neq s5.\textit{id} \\
s1.\textit{position} = \textit{camera}.\textit{position} \\
\hline
\\
\textit{INIT} \\
\hline
\textit{feeder}.\textit{INIT} \\
\textit{rbc}.\textit{INIT} \\
\textit{camera}.\textit{INIT} \\
s1.\textit{INIT} \land s2.\textit{INIT} \land s3.\textit{INIT} \land s4.\textit{INIT} \land s5.\textit{INIT} \\
\hline
\\
\textit{AddPart} \mathrel{\widehat{=}} [\, \textit{newpart?} : \textit{Part} \,] \bullet \\
\quad \neg\ [\, \textit{obj} : \textit{Part} \mid \textit{obj} \in \operatorname{ran} \textit{feeder}.\textit{objects} \land \\
\qquad \textit{distance}(\textit{obj}.\textit{position}, \textit{feeder}.\textit{start\_of\_belt}) < \\
\qquad\quad \textit{constant\_distance} \,] \land \\
\quad \textit{feeder}.\textit{Place}[\textit{newpart?}/\textit{newobj?}] \\
\textit{RecognizePart} \mathrel{\widehat{=}} [\, \exists\, \textit{obj} : \textit{Part} \bullet \textit{obj} \in \operatorname{ran} \textit{feeder}.\textit{objects} \land \\
\quad \textit{obj}.\textit{position} = \textit{camera}.\textit{position} \,] \bullet \\
\qquad s1.\textit{ChangeStatus} \land \textit{feeder}.\textit{Stop} \land \\
\qquad (\textit{camera}.\textit{Recognize}[\textit{obj}/\textit{part?}] \bullet [\, \textit{rbc}.\textit{parttype} = \textit{type!} \,]) \\
\hline
\end{array}
$$

Figure 7. Specification for Parts Recognition Subsystem

$$
\begin{array}{l}
\textit{PartsRecognitionSubsystem(Contd...)} \\
\quad \textit{PartArrived} \\
\quad obj? : Part \\
\quad \hline \\
\quad obj?.position = rbc.rotary\_belt.start\_of\_belt \\
\quad s5.status = ON \\
\\
DivertPart\_RBC \mathrel{\widehat=} PartArrived \bullet \\
\qquad [\, parttype = obj?.type = CUP \land s2.status = OFF \,] \land \\
\qquad ([\, dir? = ANTI\_CLOCKWISE \,] \bullet rbc.RotateNinety ⨾ \\
\qquad rbc.Start\_Belts\_RBC ⨾ rbc.Move\_Belts\_RBC ⨾ \\
\qquad (rbc.Stop\_Belts\_RBC \land s2.ChangeStatus)) \\
DivertPart\_RBD \mathrel{\widehat=} PartArrived \bullet \\
\qquad [\, parttype = obj?.type = DISH \land s3.status = OFF \,] \land \\
\qquad rbc.Start\_Belts\_RBD ⨾ rbc.Move\_Belts\_RBD ⨾ \\
\qquad (rbc.Stop\_Belts\_RBD \land s3.ChangeStatus) \\
DivertPart\_RBR \mathrel{\widehat=} PartArrived \bullet \\
\qquad [\, parttype = OTHER \lor \\
\qquad\qquad (parttype = CUP \land s2.status = ON) \lor \\
\qquad\qquad (parttype = DISH \land s3.status = ON) \,] \land \\
\qquad ([\, dir? = CLOCKWISE \,] \bullet rbc.RotateNinety ⨾ \\
\qquad rbc.Start\_Belts\_RBR ⨾ rbc.Move\_Belts\_RBR ⨾ \\
\qquad (rbc.Stop\_Belts\_RBR \land s4.ChangeStatus)) \\
DiverPart \mathrel{\widehat=} DivertPart\_RBC \; [] \; DivertPart\_RBD \; [] \\
\qquad\qquad DivertPart\_RBR \\
ResetBelts \mathrel{\widehat=} s5.ChangeStatus \land \\
\qquad ([\, rbc.tablepos = NINETY \,] \Rightarrow \\
\qquad\qquad [\, dir? = CLOCKWISE \,] \bullet rbc.RotateNinety) [] \\
\qquad ([\, rbc.tablepos = TWO\_SEVENTY \,] \Rightarrow \\
\qquad\qquad [\, dir? = ANTI\_CLOCKWISE \,] \bullet rbc.RotateNinety)
\end{array}
$$

Figure 8. Specification for *Parts Recognition Subsystem (Contd...)*

The state invariant of this class asserts the following: (i) The camera and sensor $s1$ are situated at the middle of the feeder belt. The length of the feeder belt is twice the *constant_distance*, the distance to be maintained between parts. (ii) Parts on the feeder belt are separated exactly by *constant_distance*. (iii) No part appears on more than one belt. (iv) If the rotary belt is at *ZERO* position, then its front end coincides with the tail end of the feeder belt. (v) All sensors have unique i.d. numbers.

The initial condition for this class includes the initial conditions of the belts and the condition that all sensors are OFF.

The *AddPart* operation invokes *Place* in the feeder belt subject to the condition that no other part appears between the front end of the feeder belt and the camera.

The *RecognizePart* operation is a sequential composition of several operations. At first, it checks whether or not there is a part in front of the camera. If there is a part, sensor $s1$ is turned on, the feeder is stopped and the operation *Recognize* in camera is invoked. The *Recognize* operation sets the shared variable *parttype* (shared between rotary belt control and camera) in the rotary belt control.

The operation *DivertPart* nondeterministically diverts a part to one of the channels. The three divert operations $DivertPart\_RBC$, $DivertPart\_RBD$ and $DivertPart\_RBR$ specify the constraints to move a part to the appropriate channel when the part arrives at the front end of the rotary belt. All the three divert operations have similar behavior: rotate the rotary belt, if necessary, in the appropriate direction, start the rotary belt and the channel belt, move both the belts and stop the belts when the part reaches the tail end of the channel. The sensor at the tail end of the channel is set when the part arrives there. Since the precondition is common for all the three operations, the precondition is separately specified in *PartArrived* which is conjoined with the three divert operations.

Notice the condition for rotating the rotary belt: if the part is a cup and there is no cup waiting for pickup at the end of *ChanC*, then the rotary belt is rotated and aligned with *ChanC*. A similar explanation can be given to move a dish along *ChanD*. If the part is a neither a cup nor a dish, or it is a cup but there is another cup waiting for pick up at the end of *ChanC*, or it is a dish and there is another dish waiting at the end of *ChanD*, then the part is diverted to *ChanR*.

The operation *ResetBelts* resets the sensor S5 at the front end of the rotary belt and then rotates the belt so that it comes back to *ZERO* position again.

The *Assembly Subsystem* specification is shown in Figure 9. The specification asserts in a formal declarative way the correct sequence of operations leading to an assembly. This is obtained by composing the specifications of robot arms, the assembly pad, the rotary table, *ChanC*, *ChanD* and the sensors $s2$ and $s3$. The channels and sensors are the communicating variables between the assembly subsystem and the parts recognition subsystem.

$$
\begin{array}{l}
\textit{AssemblySubsystem} \\
\hline
\quad chanC, chanD : Belt \\
\quad s2, s3 : Sensor \\
\quad left, right : RobotArm \\
\quad assempad : AssemblyPad \\
\quad rt : RotaryTable \\
\hline
\quad \text{INIT} \\
\quad chanC.INIT \land chanD.INIT \\
\quad s2.INIT \land s3.INIT \\
\quad left.INIT \land right.INIT \\
\quad assempad.INIT \land rt.INIT \\
\quad left.position = leftarm\_initial\_position \\
\quad right.position = rightarm\_initial\_position \\
\hline
PickupCup \mathrel{\widehat=} [\, s2.status = ON \,] \Rightarrow \\
\qquad [\, \exists\, obj : Part \bullet obj.position = chanC.end\_of\_belt \,] \bullet \\
\qquad left.Pick[obj/obj?] \land s2.ChangeStatus \\
PickupDish \mathrel{\widehat=} [\, s3.status = ON \,] \Rightarrow \\
\qquad [\, \exists\, obj : Part \bullet obj.position = chanD.end\_of\_belt \,] \bullet \\
\qquad right.Pick[obj/obj?] \land s3.ChangeStatus \\
PlaceCup \mathrel{\widehat=} [\, assempad.status = DISH\_IS\_THERE \,] \land \\
\qquad left.Place \bullet ([\, newst : PartStatus \mid \\
\qquad\qquad newst = ON\_ASSEMBLY\_PAD \,] \bullet \\
\qquad obj.ChangeStatus[newst/newst?] \land \\
\qquad [\, assempad.status' = PAIR\_IS\_THERE \,] \\
PlaceDish \mathrel{\widehat=} [\, assempad.status = NOTHING \,] \land \\
\qquad right.Place \bullet [\, newst : PartStatus \mid \\
\qquad\qquad newst = ON\_ASSEMBLY\_PAD \,] \bullet \\
\qquad obj.ChangeStatus[newst/newst?] \land \\
\qquad [\, assempad.status' = DISH\_IS\_THERE \,] \\
Assemble \mathrel{\widehat=} assempad.PerformAssembly \bullet [\, rt.done = result! \,] \\
RemoveAssembly \mathrel{\widehat=} [\, \exists\, ap : AssembledPart \bullet \\
\qquad\qquad ap.position = assempad.position \,] \bullet \\
\qquad rt.Rotate[ap/original?] \\
\hline
\end{array}
$$

Figure 9. Specification for the class *AssemblySubsystem*

The operation *PickupCup* checks to ensure that there is a cup in front of sensor $s2$ (also indicated by the presence of a part at the tail end of channel C), invokes the pick up operation by the left arm and resets the sensor $s2$. Similarly, the pick up operation for dish is also defined at the subsystem level. The *PlaceCup* operation checks whether or not there is a dish on the assembly pad (indicated by the status of the assembly pad) and if there is a dish it invokes the place operation by the left arm. The *PlaceDish* operation places a dish on the assembly pad only if the assembly pad is free of any part. Both place operations change the status of the assembly pad appropriately. The operation *Assemble* uses *PerformAssembly* operation in *assempad* (of type *AssemblyPad*) and also sets the shared variable *done* in *rt* (of type *RotaryTable*). Removing the assembled part is performed by the operation *Rotate* in *rt*.

5.2. FILTER SPECIFICATION

In this section, we give filter specifications corresponding to the Object-Z specifications. A filter specification consists of a set of filters. There is exactly one filter specification corresponding to each class and there is exactly one filter for each operation in a class. The filter is specified using the box style if it corresponds to an operation specified using the same style. If an operation is defined using composition operators, then its corresponding filter is also specified using the same set of composition operators and the filters corresponding to the operands. For an example, an operation OP expressed as

$$OP \mathrel{\widehat{=}} OP_1 \wedge OP_2$$

requires its filter express timing constraints using the semantics [2]. Accordingly, the invocation time t of OP is propagated to OP_1 and OP_2 to determine their invocation times; the anticipated completion time of OP is determined from the completion times of OP_1 and OP_2.

5.2.1. *Global Timing Variables*

Timing variables and constants derived from the problem description are shown in Figure 10; these are declared as global constants so that all filter specifications can refer to them. The constraints given in Section 4.3 are also captured in the filter specifications.

5.2.2. *Filter Specifications and Filters*

Filter specifications corresponding to *Parts Recognition Subsystem* and *Assembly Subsystem* are given below:

The filter specification corresponding to *Parts Recognition Subsystem* is described in Figures 11–13. The filter for the initial state asserts that all components of the subsystem must be initialized at the same time when the subsystem is initialized.

$$
\begin{array}{|l}
s : \mathbb{R} \\
d : \mathbb{R} \\
t_r, t_s, t_d, t_{pi}, t_{pl}, t_{ro}, t_{ao}, t_{as}, t_{fb} : Time \\
\hline
t_d = d / s \\
t_s > t_r \\
t_s \geq t_d + 2 * t_{ro} \\
t_{fb} < t_{ro} + 2 * t_d
\end{array}
$$

Figure 10. Timing Variables and Constants

$PartsRecognitionSubsystem$ **filter**

$\textsc{Init}$

$$
\begin{array}{|l}
feeder : Belt \\
rbc : RotaryBeltChannels \\
camera : Camera \\
s1, s2, s3, s4, s5 : Sensor \\
t : RealTime \\
\hline
s1.INIT[t / s1.INIT.t] \land s2.INIT[t / s2.INIT.t] \land \\
s3.INIT[t / s3.INIT.t] \land s4.INIT[t / s4.INIT.t] \land \\
s5.INIT[t / s5.INIT.t] \\
feeder.INIT[t / feeder.INIT.t] \land rbc.INIT[t / rbc.INIT.t] \\
camera.INIT[t / camera.INIT.t]
\end{array}
$$

$$
\begin{array}{l}
AddPart \mathrel{\widehat{=}} [\, mapping = \{newpart? \mapsto t + \epsilon_{1\,newpart?}, feeder \mapsto t\} \,] \land \\
\quad feeder.Place[newpart? / newobj?, t / feeder.Place.t]
\end{array}
$$

$$
\begin{array}{l}
RecognizePart \mathrel{\widehat{=}} [\, mapping = \{feeder \mapsto t, camera \mapsto t, rbc \mapsto t\} \,] \land \\
\quad s1.ChangeStatus[t / s1.ChangeStatus.t] \land \\
\quad feeder.Stop[t / feeder.Stop.t] \land \\
\quad (camera.Recognize[obj / part?, t / camera.Recognize.t])
\end{array}
$$

$PartArrived$

$$
\begin{array}{|l}
rbc : RotaryBeltChannels \\
s5 : Sensor \\
obj? : Part \\
t : RealTime \\
\delta t, \epsilon_{1\,obj?} : Time \\
\hline
mapping = \{obj? \mapsto t + \epsilon_{1\,obj?}, rbc \mapsto t, s5 \mapsto t\}
\end{array}
$$

Figure 11. Filter Specification corresponding to *Parts Recognition Subsystem*

$$
\begin{array}{l}
\textit{PartRecognitionSubsystem\_}\mathbf{filter}(\textit{Contd...}) \\
\hline
\textit{DivertPart\_RBC} \mathrel{\widehat=} \textit{PartArrived}[t/\textit{PartArrived}.t] \land \\
\quad [\, \textit{mapping} = \{ \textit{parttype} \mapsto t, s2 \mapsto t, \textit{dir?} \mapsto t + \epsilon_{1\,\textit{dir?}} \} \,] \land \\
\quad \textit{rbc.RotateNinety}[t/\textit{rbc.RotateNinety}.t] \land \\
\quad \textit{rbc.Start\_Belts\_RBC}[t + \textit{rbc.RotateNinety}.\delta t/ \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBC}.t] \land \\
\quad \textit{rbc.Move\_Belts\_RBC}[t + \textit{rbc.RotateNinety}.\delta t + \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBC}.\delta t/\textit{rbc.Move\_Belts\_RBC}.t] \land \\
\quad \textit{rbc.Stop\_Belts\_RBC}[t + \textit{rbc.RotateNinety}.\delta t + \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBC}.\delta t + \textit{rbc.Move\_Belts\_RBC}.\delta t/ \\
\qquad\qquad \textit{rbc.Stop\_Belts\_RBC}.t] \land \\
\quad s2.\textit{ChangeStatus}[t + \textit{rbc.RotateNinety}.\delta t + \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBC}.\delta t + \textit{rbc.Move\_Belts\_RBC}.\delta t/ \\
\qquad\qquad s2.\textit{ChangeStatus}.t] \\
\textit{DivertPart\_RBD} \mathrel{\widehat=} \textit{PartArrived}[t/\textit{PartArrived}.t] \land \\
\quad [\, \textit{mapping} = \{ \textit{parttype} \mapsto t, s3 \mapsto t \} \,] \land \\
\quad \textit{rbc.Start\_Belts\_RBD}[t/\textit{rbc.Start\_Belts\_RBD}.t] \land \\
\quad \textit{rbc.Move\_Belts\_RBD}[t + \textit{rbc.Start\_Belts\_RBD}.\delta t/ \\
\qquad\qquad \textit{rbc.Move\_Belts\_RBD}.t] \land \\
\quad \textit{rbc.Stop\_Belts\_RBD}[t + \textit{rbc.Start\_Belts\_RBD}.\delta t + \\
\qquad\qquad \textit{rbc.Move\_Belts\_RBD}.\delta t/\textit{rbc.Stop\_Belts\_RBD}.t] \land \\
\quad s3.\textit{ChangeStatus}[t + \textit{rbc.Start\_Belts\_RBD}.\delta t + \\
\qquad\qquad \textit{rbc.Move\_Belts\_RBD}.\delta t/s3.\textit{ChangeStatus}.t] \\
\textit{DivertPart\_RBR} \mathrel{\widehat=} \textit{PartArrived}[t/\textit{PartArrived}.t] \land \\
\quad [\, \textit{mapping} = \{ \textit{parttype} \mapsto t, s2 \mapsto t, s3 \mapsto t, \textit{dir?} \mapsto t + \epsilon_{1\,\textit{dir?}} \} \,] \land \\
\quad \textit{rbc.RotateNinety}[t + \epsilon_1/\textit{rbc.RotateNinety}.t] \land \\
\quad \textit{rbc.Start\_Belts\_RBR}[t + \textit{rbc.RotateNinety}.\delta t/ \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBR}.t] \land \\
\quad \textit{rbc.Move\_Belts\_RBR}[t + \textit{rbc.RotateNinety}.\delta t + \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBR}.\delta t/\textit{rbc.Move\_Belts\_RBR}.t] \land \\
\quad \textit{rbc.Stop\_Belts\_RBR}[t + \textit{rbc.RotateNinety}.\delta t + \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBR}.\delta t + \textit{rbc.Move\_Belts\_RBR}.\delta t/ \\
\qquad\qquad \textit{rbc.Stop\_Belts\_RBR}.t] \land \\
\quad s4.\textit{ChangeStatus}[t + \textit{rbc.RotateNinety}.\delta t + \\
\qquad\qquad \textit{rbc.Start\_Belts\_RBR}.\delta t + \textit{rbc.Move\_Belts\_RBR}.\delta t/ \\
\qquad\qquad s4.\textit{ChangeStatus}.t] \\
\hline
\end{array}
$$

Figure 12. Filter Specification corresponding to *Parts Recognition Subsystem* (Contd...)

$$
\begin{array}{l}
\textit{PartRecognitionSubsystem\_}\mathbf{filter}(\textit{Contd...}) \\
\hline
\textit{DivertPart} \mathrel{\widehat{=}} \textit{DivertPart\_RBC}[t / \textit{DivertPart\_RBC}.t] \wedge \\
\quad \textit{DivertPart\_RBD}[t / \textit{DivertPart\_RBD}.t] \wedge \\
\quad \textit{DivertPart\_RBR}[t / \textit{DivertPart\_RBR}.t] \\
\textit{ResetBelts} \mathrel{\widehat{=}} s5.\textit{ChangeStatus}[t / s5.\textit{ChangeStatus}.t] \wedge \\
\quad ([\, \textit{mapping} = \{\textit{rbc} \mapsto t, \textit{dir?} \mapsto t + \epsilon_{1\,\textit{dir?}}\} \,] \wedge \\
\qquad \textit{rbc}.\textit{RotateNinety}[t / \textit{rbc}.\textit{RotateNinety}.t]) \wedge \\
\quad ([\, \textit{mapping} = \{\textit{rbc} \mapsto t, \textit{dir?} \mapsto t + \epsilon_{1\,\textit{dir?}}\} \,] \wedge \\
\qquad \textit{rbc}.\textit{RotateNinety}[t / \textit{rbc}.\textit{RotateNinety}.t]) \\
\hline
\end{array}
$$

Figure 13. Filter Specification corresponding to *Parts Recognition Subsystem* (Contd...)

If an operation is defined as a composition of several other operations using the composition operators, then the corresponding filter will contain two sections: (i) The first section introduces an unnamed schema in the filter. This unnamed schema defines the mapping of the functional variables involved in the operation to their respective timing parameters. These functional variables are brought into scope through unnamed schemas in the operation; see, for example, the schema

$[\, \textit{newpart?} : \textit{Part} \,]$

in the operation *AddPart*. (ii) The second section describes how the invocation time of the composite operation is propagated to the invocation times of the component operations. The filter corresponding to the composite operation is defined as a logical conjunction of the three sections irrespective of the composition operators used in its associated operation definition.

The filter *PartArrived* is self-explanatory. The filter *DivertPart_RBC* specifies the propagation of the timing constraints through the sequence of operations which compose *DivertPart_RBC*. It consists of two sections. The first section describes the anticipated completion time of *DivertPart_RBC* which is the maximum time taken by the sequence of operations. The next section specifies the times of invocations of the sequence of operations required to complete the task. Filters for other divert operations can be interpreted in a similar fashion.

Notice that in the filter *ResetBelts*, the constraint

$[\, \textit{mapping} = \{\textit{rbc} \mapsto t, \textit{dir?} \mapsto t + \epsilon_{1\,\textit{dir?}}\} \,] \wedge$
$\textit{rbc}.\textit{RotateNinety}[t / \textit{rbc}.\textit{RotateNinety}.t]$

appears twice. This is because of the direct application of our methodology to two different set of functional constraints involving the same variables (refer to the operation *ResetBelts*). By the semantics of logical conjunction, the two constraints in the filter *ResetBelts* will be merged into a single constraint in the timed specification.

$$
\begin{array}{l}
\textit{AssemblySubsystem}\_\textbf{filter} \\
\quad \begin{array}{l}
\textit{INIT} \\
\textit{chanC}, \textit{chanD} : \textit{Belt} \\
s2, s3 : \textit{Sensor} \\
\textit{left}, \textit{right} : \textit{RobotArm} \\
\textit{assempad} : \textit{AssemblyPad} \\
\textit{rt} : \textit{RotaryTable} \\
t : \textit{RealTime} \\
\hline
\textit{chanC}.\textit{INIT}[t / \textit{chanC}.\textit{INIT}.t] \\
\textit{chanD}.\textit{INIT}[t / \textit{chanD}.\textit{INIT}.t] \\
s2.\textit{INIT}[t / s2.\textit{INIT}.t] \\
s3.\textit{INIT}[t / s3.\textit{INIT}.t] \\
\textit{left}.\textit{INIT}[t / \textit{left}.\textit{INIT}.t] \\
\textit{right}.\textit{INIT}[t / \textit{right}.\textit{INIT}.t] \\
\textit{assempad}.\textit{INIT}[t / \textit{assempad}.\textit{INIT}.t] \\
\textit{rt}.\textit{INIT}[t / \textit{rt}.\textit{INIT}.t] \\
\textit{mapping} = \{\textit{left} \mapsto t, \textit{right} \mapsto t\}
\end{array} \\
\quad \textit{PickupCup} \mathrel{\widehat=} [\, \textit{mapping} = \{s2 \mapsto t, \textit{chanC} \mapsto t\} \,] \wedge \\
\qquad \textit{left}.\textit{Pick}[\textit{obj} / \textit{obj}?, t / \textit{left}.\textit{Pick}.t] \wedge \\
\qquad s2.\textit{ChangeStatus}[t / s2.\textit{ChangeStatus}.t] \\
\quad \textit{PickupDish} \mathrel{\widehat=} [\, \textit{mapping} = \{s3 \mapsto t, \textit{chanD} \mapsto t\} \,] \wedge \\
\qquad \textit{right}.\textit{Pick}[\textit{obj} / \textit{obj}?, t / \textit{right}.\textit{Pick}.t] \wedge \\
\qquad s3.\textit{ChangeStatus}[t / s3.\textit{ChangeStatus}.t] \\
\quad \textit{PlaceCup} \mathrel{\widehat=} [\, \textit{mapping} = \{\textit{assempad} \mapsto t, \textit{assempad}' \mapsto t + \delta t\} \,] \wedge \\
\qquad \textit{left}.\textit{Place}[t / \textit{left}.\textit{Place}.t] \\
\quad \textit{PlaceDish} \mathrel{\widehat=} [\, \textit{mapping} = \{\textit{assempad} \mapsto t, \textit{assempad}' \mapsto t + \delta t\} \,] \wedge \\
\qquad \textit{right}.\textit{Place}[t / \textit{right}.\textit{Place}.t] \\
\quad \textit{Assemble} \mathrel{\widehat=} [\, \textit{mapping} = \{\textit{rt} \mapsto t\} \,] \wedge \\
\qquad \textit{assempad}.\textit{PerformAssembly}[t / \textit{assempad}.\textit{PerformAssembly}.t] \\
\quad \textit{RemoveAssembly} \mathrel{\widehat=} [\, \textit{mapping} = \{\textit{assempad} \mapsto t\} \,] \\
\qquad \textit{rt}.\textit{Rotate}[\textit{ap} / \textit{original}?, t / \textit{rt}.\textit{Rotate}.t]
\end{array}
$$

Figure 14. Filter Specification corresponding to *AssemblySubsystem*

The filter specification corresponding to the class *Assembly Subsystem* is given in Figure 14. The filter *PickupCup* has an anticipated completion time which is the maximum of the anticipated completion time of *Pick* operation in *left* and that of *ChangeStatus* in sensor $s2$. The values of the variables $s2$ and $chanC$ in the predicate part of the operation *PickupCup* must be interpreted according to their respective values known at the time of invocation of *PickupCup* and hence these variables are mapped to t in the filter *PickupCup*. If the precondition is satisfied, the operation *Pick* in *left* and the *ChangeStatus* in $s2$ will be invoked at time t.

Other filter specifications in *Assembly Subsystem*_**filter** can be understood in a similar way.

5.3. APPLYING FILTER SPECIFICATIONS TO CLASSES

In this section, we illustrate how the filter specification can be applied to a class definition. This follows from the semantics of the filter specifications given in [2]. We show the application of the semantics for *AddPart* operation in Figure 15.

The application of a filter specification to its corresponding class definition will generate a timed class definition, embedding the real-time constants on the functional constraints. Each functional variable will be augmented with the timing variable to which it is mapped in the filter specification. According to the semantics [2], this application results in the value of the variable at this time. When an operation name is augmented with a time variable, the timed operation denotes the invocation time of the operation. For example, the operation *feeder.Place* in the predicate $feeder.Place_t[newpart?_t/newobj?]$ in Figure 16 is augmented with t, and should be understood to mean that *feeder.Place* is invoked at time t with the value of variable *newpart*? at time t assigned to the input variable $newobj?$.

The interpretation here is that *newpart*? is received $\epsilon_{1newpart?}$ units of time after *AddPart* is invoked. There is a precondition for this operation which asserts that no part in the feeder line should be within a distance of *constant_distance* from the start of the feeder belt at time t. If the assertion holds, the operation *Place* will be invoked at time t.

6. Verification

In this section, we state safety and liveness properties and give rigorous proofs for the satisfaction of these properties in the assembly cell specification.

$$
\begin{array}{|l}
\hline
\textit{PartsRecognitionSubsystem} \\
\ldots \\
AddPart \mathrel{\widehat=} [\, newpart?_{t+\epsilon_{1newpart?}} : Part \,] \bullet \\
\quad \neg \; [\, obj_t : Part \mid obj_t \in \operatorname{ran} feeder.objects_t \wedge \\
\qquad distance(obj.position_t, feeder.start\_of\_belt_t) < \\
\qquad\quad constant\_distance \,] \wedge \\
\quad feeder.Place_t[newpart?_t / newobj?] \\
\ldots \\
\hline
\end{array}
$$

Figure 15. Real-Time Object-Z Specification for Parts Recognition Subsystem

6.1. SAFETY PROPERTY

Property 1: No collision occurs between parts when they are on belts.

There are two situations:
1.1 Parts on the same belt.
The state invariant of the class *Belt* is that no two parts on any belt can occupy the same position. This is asserted by the predicate

$\forall i, j : 1 \mathrel{..} \#objects \bullet i \neq j \Rightarrow object(i).position \neq object(j).position$

Thus, in the specification the property holds.

1.2 Parts on different belts.
A collision might occur between two parts p_1 and p_2 when p_1 is on a belt b_1 and p_2 is on another belt b_2. This is possible only when the start of belt b_2 coincides with the end of belt b_1 or vice versa. From the system configuration, this situation occurs only between rotary belt and any of the three channel belts or between the rotary belt and the feeder.

Case 1: p_1 is on feeder line and p_2 is on rotary belt.
A collision occurs only if p_2 is stationary and p_1 is moving. From the timed specification for *Parts Recognition Subsystem*, the operation *DivertPart* will be invoked at the same time when a part arrives in front of sensor $s5$. This causes the rotary belt to rotate and align with one of the channels and initiates the motion of the rotary belt and the appropriate channel belt. Therefore, p_1 cannot collide with p_2. Further the rotary belt will reset and align itself with the feeder before the next part arrives from the feeder. This is ensured by the timing constraint $t_s \geq 2 * t_{ro} + t_d$. Therefore, p_1 will not slip from the feeder.

Case 2: p_1 is on rotary belt and p_2 is on one of the channel belts.
Let us assume without loss of generality that p_2 is on *ChanC*. From the timed specification for the operation *DivertPart_RBC* in *Parts Recognition Subsystem*, the rotary belt and *ChanC* are started at the same time. Further, they also move at the same time and at the same speed. Because their lengths are equal, (all the belts have same characteristics) the move operations will be completed at the same time. Moreover, if there is already a cup in *ChanC*, then according to our specification, rotary belt will align with *ChanR* and return the cup on the rotary belt through *ChanC*.

Therefore, there will be no collision between parts when they are on belts.

Property 2: No two parts of the same kind are placed on the assembly pad.

The proof is a straightforward consequence of the preconditions for the operations *PlaceCup* and *PlaceDish* in the timed specification for *Assembly Subsystem*.

6.2. LIVENESS PROPERTY

The robotics assembly system, as specified, must achieve the intended tasks stated in the requirements. In particular, all assemblies of parts must be completed in bounded time. We state this as a liveness property.

If k cups and k dishes are placed on the feeder then irrespective of the ordering of the cups and dishes, all k assemblies will be correctly completed within a finite time.

To prove this property, we first prove the following property:

If k cups and k dishes are placed on the feeder then irrespective of the ordering of the cups and dishes, the first assembly will be made within the time

$$s_{k+1} + t_r + t_d + t_{ro} + 2 * t_d + t_{pi} + 2 * t_{pl} + t_{as} + t_{ao}$$

where s_i denotes the time at which the i^{th} part (cup or dish) reaches the camera.

We determine the maximum time taken to assemble the first pair of cup and dish given the worst case scenario. The worst case scenario for this problem will be the situation when all cups are fed first followed by all dishes. The first assembly takes the maximum time, since the robot has to wait until the first dish arrives.

Let $f(k)$ be the maximum time taken for one assembly when k cups and k dishes are fed, in that order. We recall the following assumptions stated as part of the requirements:

- A part is placed only at the start of the feeder belt.
- When a part is placed on the feeder, there is no other part between the camera and the start of the feeder.
- The next part, if available, is fed when the previous part reaches the camera. In other words, there is no delay in feeding.
- The time at which the entire assembly process is initiated is measured from the time at which the first part is placed on the feeder (excluding its placing time). This initial time is assumed to be 0.

In order to calculate the time at which the first assembly is made, we must first calculate the time at which the first dish appears in front of the camera and then add the durational delays for subsequent actions such as recognition, move to *ChanD*, pick and place. Due to the overlap of actions and of our assumption that all cups are fed before the first dish, the time taken for the first cup to be picked up by the left arm will be included in the time taken for the first dish to be picked up by the right arm.

Let s_i denote the time at which the i^{th} part (cup or dish) arrives at the camera. The i^{th} part is placed on the feeder when the $(i-1)^{th}$ part is being recognized. So, the time at which the i^{th} cup reaches the camera is the sum of (1) the time at which the $(i-1)^{th}$ cup, reached the camera, (2) the time taken for the $(i-1)^{th}$ cup to be recognized (i.e., t_r), and (3) the time taken for the i^{th} cup to travel from the start of the feeder to the camera (i.e., t_d). A partial event chart shown in Figure 16 illustrates this scenario.

Hence,

$$s_i = s_{i-1} + t_r + t_d\,,\; s_1 = t_d$$

Thus, s_k, the time at which the k^{th} cup reaches the camera is

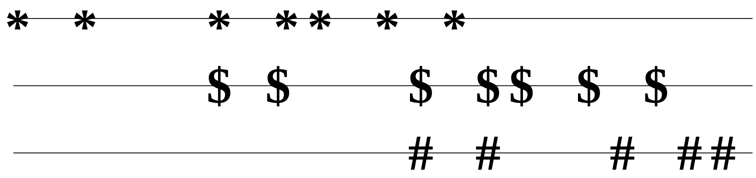
$$s_k = s_{k-1} + t_r + t_d$$

Since the first dish follows the k^{th} cup, the time at which the first dish reaches the camera is

$$s_{k+1} = s_k + t_r + t_d$$

To the time of arrival of the first dish at the camera (i.e., s_{k+1}), we can now add the following times to calculate the time taken for the first assembly:

t_r	-	to recognize the dish
t_d	-	to travel from the camera to the rotary belt
t_{ro}	-	to rotate the rotary belt



Legend:

*** denotes the occurrences of events for the first cup**
$ denotes the occurrence of events for the second cup
 # denotes the occurrences of events for the third cup

The first occurrence denotes "Place"
The second occurrence denotes "Start of recognition"
The third occurrence denotes "End of recognition"
The fourth occurrence denotes "Enter rotary belt"
The fifth occurrence denotes "Rotate the rotary belt"
The sixth occurrence denotes "Move cup across rotary belt"
The seventh occurrence denotes "Move cup across channel C or channel R"

Figure 16. Events Chart for the Assembly Process

$2 * t_d$ - to travel from the start of the rotary belt to end of *ChanD*
t_{pi} - to be picked up by the right arm
t_{pl} - to place the dish on the assembly pad
t_{pl} - to place the cup on the assembly pad
t_{as} - to assemble the pair
t_{ao} - to rotate and remove the assembled pair

Notice that the duration for the first dish to arrive at the assembly pad includes the duration for the first cup to be picked up by the left arm.

The maximum time taken for the first assembly to be made is
$f(k) = s_{k+1} + t_r + 3 * t_d + t_{ro} + t_{pi} + 2 * t_{pl} + t_{as} + t_{ao}$
Let us informally verify that this expression is true for k = 1; i.e., one cup followed by one dish. The dish is placed when the cup is recognized by

the camera. Thus, the dish appears at time $t_r + t_d + t_d$ units after the system started. That is, $s_2 = t_r + 2 * t_d$. The time taken for the first assembly is

$$\begin{aligned} & s_2 + t_r + 3 * t_d + t_{ro} + t_{pi} + 2 * t_{pl} + t_{as} + t_{ao} \\ = \ & 2 * t_r + 5 * t_d + t_{ro} + t_{pi} + 2 * t_{pl} + t_{as} + t_{ao} \end{aligned}$$

which is $f(1)$. The factor $2 * t_r$ includes the recognition time for cup and dish. The two recognition times are cumulative because the cup is recognized before the dish is placed. The factor $5 * t_d$ denotes the net cumulative time taken by the dish to travel from the start of the feeder belt to the end of *ChanD* and the time taken by the cup to travel from the start of feeder to the end of *ChanC*. The factor t_{ro} represents the time taken for the rotary belt to rotate and t_{pi} denotes the pick up time for the dish. The rotation time and the pick up time for the cup is subsumed by the time for the dish to appear at the end of *ChanD*. The factor $2 * t_{pl}$ denotes the cumulative placing time for the cup and the dish. The factors t_{as} and t_{ao} denote the assembly time and the removal time of the assembled part.

6.2.1. *An Assembly Algorithm from Specifications*

An algorithm to perform the assembly using the objects in the specification is given in Figure 17. It is our goal to compute the anticipated completion time of this algorithm from the timed specification and show that for assembling one cup and one dish the result is $f(1)$, matching the calculated time in the previous section. We use the abbreviations *PRS* and *ASS* to denote *Parts Recognition Subsystem* and *Assembly Subsystem* respectively.

We prove now that this algorithm makes one assembly in time $f(1)$. We first compute the anticipated completion time for each step separately and then add all of them. In the computations below, we use the semantics of operator compositions for the real-time specifications [2]. For example, the anticipated completion time of Op in the expression $Op \mathrel{\widehat{=}} Op_1 \wedge Op_2$ is the maximum of the anticipated completion times of Op_1 and Op_2. Similarly, the anticipated completion time of Op in the expression $Op \mathrel{\widehat{=}} Op_1 \mathbin{;} Op_2$ is that sum of anticipated completion times of Op_1 and Op_2.

$$\begin{aligned} PRS.AddPart.\delta t &\simeq PRS.feeder.Place.\delta t \\ &\simeq 0 \end{aligned}$$

$$PRS.feeder.Start.\delta t \simeq 0$$

$$\begin{aligned} PRS.feeder.Move.\delta t &= constant\_distance/s \\ &= t_d \end{aligned}$$

Input: One cup followed by one dish.
Output: An Assembled Part.
Algorithm:

$$
\begin{array}{l}
PRS.AddPart[Cup/newpart?]\,\S\\
PRS.feeder.Start\,\S\\
PRS.feeder.Move[constant\_distance/dist?]\,\S\\
(PRS.RecognizePart \wedge PRS.AddPart[Dish/newpart?])\,\S\\
PRS.feeder.Start\,\S\\
PRS.feeder.Move[constant\_distance/dist?]\,\S\\
(PRS.DivertPart\_RBC\,\S\\
\quad (PRS.Resetbelts \wedge ASS.PickupCup)\\
)\wedge\\
(PRS.RecognizePart\,\S\\
\quad PRS.feeder.Start\,\S\\
\quad PRS.feeder.Move[constant\_distance/dist?]\,\S\\
\quad PRS.DivertPart\_RBD\,\S\\
\quad (PRS.ResetBelts \wedge ASS.PickupDish)\\
)\,\S\\
ASS.PlaceDish\,\S\\
ASS.PlaceCup\,\S\\
ASS.Assemble\,\S\\
ASS.RemoveAssembly
\end{array}
$$

Figure 17. An Assembly Algorithm

$$
\begin{aligned}
PRS.RecognizePart.\delta t &= max(PRS.s1.ChangeStatus.\delta t, PRS.feeder.Stop.\delta t,\\
&\qquad PRS.camera.Recognize.\delta t)\\
&= PRS.camera.Recognize.\delta t\\
&= t_r\\
PRS.DivertPart\_RBC.\delta t &= max(PRS.PartArrived.\delta t,\\
&\qquad PRS.rbc.RotateNinety.\delta t +\\
&\qquad PRS.rbc.Start\_Belts.\_RBC.\delta t +\\
&\qquad PRS.rbc.Move\_Belts\_RBC.\delta t,\\
&\qquad PRS.s2.ChangeStatus.\delta t)\\
&= PRS.rbc.RotateNinety.\delta t +\\
&\quad PRS.rbc.Start\_Belts.\_RBC.\delta t +\\
&\quad PRS.rbc.Move\_Belts\_RBC.\delta t\\
&\simeq t_{ro} + 0 + 2 * t_d\\
&\simeq t_{ro} + 2 * t_d\\
PRS.ResetBelts.\delta t &= max(PRS.s5.ChangeStatus.\delta t,\\
&\qquad PRS.rbc.RotateNinety.\delta t)\\
&= PRS.rbc.RotateNinety.\delta t\\
&= t_{ro}
\end{aligned}
$$

$$
\begin{aligned}
ASS.PickupCup.\delta t &= max(ASS.left.Pick.\delta t, \\
&\qquad ASS.s2.ChangeStatus.\delta t) \\
&= ASS.left.Pick.\delta t \\
&= t_{pi}
\end{aligned}
$$

$PRS.DivertPart\_RBD.\delta t = t_{ro} + 2 * t_d$

This is the same as $PRS.DivertPart\_RBC.\delta t$.

$ASS.PickupDish.\delta t = t_{pi}$

This is the same as $ASS.PickupCup.\delta t$.

$$
\begin{aligned}
ASS.PlaceDish.\delta t &= ASS.right.Place.\delta t \\
&= t_{pl}
\end{aligned}
$$

$ASS.PlaceCup.\delta t = t_{pl}$

This is the same as $ASS.PlaceDish.\delta t$.

$$
\begin{aligned}
ASS.Assemble.\delta t &= ASS.assempad.PerformAssembly.\delta t \\
&\simeq t_{as}
\end{aligned}
$$

$$
\begin{aligned}
ASS.RemoveAssembly.\delta t &= ASS.rt.Rotate.\delta t \\
&\simeq t_{ao}
\end{aligned}
$$

The time taken for the first assembly is the sum of time taken for the steps in the assembly algorithm. That is,

$$
\begin{aligned}
& 0 + 0 + t_d + max(t_r, 0) + 0 + t_d + max(t_{ro} + 2 * t_d + max(t_{pi}, t_{ro}), \\
& t_r + 0 + t_d + t_{ro} + 2 * t_d + max(t_{pi}, t_{ro})) + t_{pl} + t_{pl} + t_{as} + t_{ao} \\
= \; & t_r + 2 * t_d + max(t_{ro} + 2 * t_d + t_{pi}, t_r + t_{ro} + 3 * t_d + t_{pi}) + \\
& 2 * t_{pl} + t_{as} + t_{ao} \\
= \; & 2 * t_r + 5 * t_d + t_{ro} + t_{pi} + 2 * t_{pl} + t_{as} + t_{ao} \\
= \; & f(1)
\end{aligned}
$$

7. Conclusion

An intelligent control must behave as expected without failure during its life cycle. This mandates the designer of the control to establish the correctness of control software prior to installing the control in practical systems. In principle, this requires a proof of correspondence between the control system's specification and the software itself (say, written in a high level language). In this chapter we have provided a rigorous justification to establish that the (design) specification of the control implements the requirements specification. That is, the rigorous proof is the off-line verification of the specified control system.

A system can be verified (even if the process is only informal) only if its behavior is formally expressed using some mathematical notation. We have chosen Object-Z, a formal specification language to specify the control of a robotic assembly system. The Object-Z notation achieves modularity in design. Due to this reason the specification is easy to read, understand, and reason about.

To express formally the real-time constraints on processes and the data they consume, we have introduced filters having the same syntax and semantics as classes in Object-Z. The novelty of this approach is that functional aspects and real-time constraints are decoupled at the specification level. With this design both expressiveness of state abstractions and modularity in timing specifications are obtained. Consequently, the resulting design is reusable in contexts where timing properties are not insisted. Inheritance hierarchies can be defined independently in function description modules and timing constraint modules to support multiple applications where different timing restrictions on the same functionality (timing constraints) are demanded.

In [10], several specification notations have been employed to specify a similar problem. None of these methods seem to tackle both design complexity and formal verification. We have given in [2] a logic for Real-time Object Z, which also gives an abstract axiomatic framework for formally verifying the timed specifications obtained by an application of filters to class specifications. Based on this logic, an exact analysis and formal verification of control systems specified in our methodology are feasible, although it may be prohibitively expensive. However, formal analysis may lead to system improvements. For example, we can derive an expression for t_{pi} for efficient utilization of the system.

An efficient utilization of the system results when two cups are placed in sequence and the left arm is initially empty, the second cup will be diverted to channel C. This happens if and only if the first cup is picked up by the left arm at or before the second cup arrives at the front end of the rotary belt. Remember that the decision to rotate the rotary belt towards appropriate channel is taken only when the part arrives at the rotary belt. From the specifications of *Assembly Subsystem* and *Parts Recognition Subsystem*, this can be stated as:
The task

$$PRS.DivertPart\_RBC \fatsemi (PRS.ResetBelts \wedge ASS.PickupCup)$$

must be completed before the task

$$PRS.RecognizePart \fatsemi PRS.feeder.Start \fatsemi \\ \quad PRS.feeder.Move[constant\_distance / dist?]$$

Let $S1 = PRS.DivertPart\_RBC \fatsemi$
$\quad (PRS.ResetBelts \wedge ASS.PickupCup)$

and

$$S2 = PRS.RecognizePart \fatsemi PRS.feeder.Start \fatsemi PRS.feeder.Move[constant\_distance/dist?]$$

From the timing expressions derived in section 6.2.1, we have

$$S1.\delta t = t_{ro} + 2 * t_d + max(t_{ro}, t_{pi})$$
$$S2.\delta t = t_r + t_d$$

For the above property to hold,

$$t_{ro} + 2 * t_d + max(t_{ro}, t_{pi}) \leq t_r + t_d$$
i.e., $$t_{ro} + t_d + max(t_{ro}, t_{pi}) \leq t_r$$

Assuming that $t_{pi} > t_{ro}$, the above expression reduces to

$$t_{ro} + t_d + t_{pi} \leq t_r$$
i.e., $$t_{pi} \leq t_r - t_{ro} - t_d$$

Adjusting the belt speed and the speed of rotation of the rotary belt, this timing expression can be made true and hence the system can be utilized efficiently.

8. Acknowledgements

This work was supported by grants from Natural Sciences and Engineering Research Council, Canada and the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche, Quebec, Canada.

References

1. V.S. Alagar and K. Periyasamy, "A Real-Time Object-Oriented Specification of a Robot-Based Assembly Cell", Technical Report, Department of Computer Science, Concordia University, Montreal, Canada, April 1996.
2. V.S. Alagar and K. Periyasamy, "A Logic for Real-Time Object-Z", Technical Report, Department of Computer Science, Concordia University, Montreal, Canada, April 1996.
3. P.J. Antsaklis and K.M. Passino (Eds.), *An Introduction to Intelligent and Autonomous Control*, Kluwer Academic Publishers, 1993.
4. M. Aksit *et al.*, "Real-Time Specification Inheritance Anomalies and Real-Time Filters", *ECOOP'94: European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science - 821, Springer-Verlag, 1994, 386–405.
5. G. Booch, *Object-Oriented Analysis and Design with Applications*, (Second Edition), Benjamin Cummings Publications, 1994.
6. R. Duke *et al.*, "Object-Z: A Specification Language Advocated for the Description of Standards", Technical Report 94-45, Department of Computer Science, University

of Queensland, Australia, Dec 1994.

7. A. Griffiths, "From Object-Z to Eiffel: Towards a Rigorous Development Method", *Proceedings of the 18th International Conference on Object-Oriented Languages and Systems (TOOLS'95)*, C. Mingins, R. Duke and B. Meyer (Eds.), Prentice Hall, Nov 1995, pp. 293–307.
8. C. Heitmeyer and N. Lynch, "The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems", *Proceedings of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec 1994, pp.120–131.
9. *IEEE Software*, Jan 1994.
10. C. Lewerentz and T. Lindner (Eds.), *Formal Development of Reactive Systems*, Lecture Notes in Computer Science - 891, Springer-Verlag, 1995.
11. B.. Meyer, *Object-Oriented Software Construction*, Prentice Hall International Series in Computer Science, 1988.
12. K. Periyasamy, "Robotic Agents and Assembly Process: A Formal Specification Case Study", Ph.D. Thesis, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, Aug 1991.
13. J. Rumbaugh *et al.*, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.
14. S.G. Tzafestas (Ed.),*Intelligent Robot Systems*, Marcel Dekker Inc., NY, 1991.