Real-Time Object-Z: A Language for the Specification and Design of Real-Time Reactive Systems[†]

V.S. Alagar
Department of Computer Science
Concordia University
1455 de Maisonneuve Blvd. West
Montreal, Quebec, Canada H3G 1M8
Ph: (514) 848-3022
Fax: (514) 848-2830
email: alagar@cs.concordia.ca

K. Periyasamy
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
Ph: (204) 474-8682
Fax: (204) 269-9178
email: kasi@cs.umanitoba.ca

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

Abstract

Models for traditional real-time systems focus either on process-oriented timing constraints or data sensitive real-time constraints. Real-time reactive systems, due to their continuous interaction with their environment, have to deal with both time dependent data and time constrained processes. To increase potential reuse and concurrency and to avoid undesirable consequences during the use of a real-time system, a design that decouples functionalities of processes from their time dependence and from the time constrained data that they consume is preferred. We have introduced a new specification language called Real-Time Object-Z (RTOZ) in this report. RTOZ is a real-time extension of the specification language Object-Z in which the functional and timing consraints are specified independently. The semantics of RTOZ, built on top of the semantics of Object-Z, describes a mapping between the functional and timing constraints which enables the derivation of a timed specification structure. A logic for RTOZ, described in this report, provides a set of proof rules supporting formal verification of real-time systems specified in RTOZ.

Contents

1 Introduction

Real-time reactive systems modeling real-world applications such as telephony, process control systems, and traffic control are inherently complex due to their continuous interaction with the environment. Consequently, modeling real-time systems is a challenging process. A reactive system typically should respect liveness and safety requirements. Safety requirements are invariant properties in the system and must be met. Liveness requirements take the form of system properties that can be observed in bounded time. Models for real-time reactive systems should enable the verification of safety and liveness properties. In addition, the modeling methodology should incorporate sound design principles so as to contain complexity at the design level.

Models for traditional real-time systems focus either on system processes and their timing characteristics or the data to be processed and its time dependent properties. Real-time reactive systems, due to their continuous interaction with the environment, have to deal with both time dependent data and time constrained processes. Most of the approaches to modeling real-time reactive systems either focus on design aspects or on the verification aspects but not on both. We present a formal methodology which reduces design complexity, promotes modular and incremental design, and facilitates formal verification. To increase potential reuse and concurrency and to avoid undesirable consequences due to unsafe executuion paths, we have decoupled the functional specification part from the time constraints, individually specified them within the same formal notation, and have given a semantics for deriving the timed specification from these specifications.

The formal framework of our methodology is Object-Z specification language. The Object-Z notation [4] does not support real-time specifications. Therefore, we have extended the Object-Z notation and semantics in a conservative manner to express timing constraints. This extended specification is called Real-Time Object-Z (RTOZ).

A specification in Object-Z has two tiers: one tier for functionalities and the other for timing constraints. Both tiers use the syntax and semantics of Object-Z. The tier where timing constraints are stated introduces *filter specifications.* The logic described in Section 2 provides an abstract, axiomatic semantics for combining the specifications of a real-time system in the two tiers to obtain the timed specification of the real-time system.

A filter specification is a collection of filters presented as schemas. A filter schema defines an association between variables (data) and real-time. Additional time constraints from problem domain may also be stated in a filter schema.

The structure of a filter specification is the same as that of a class except that it has only a set of filters. Each filter specification is associated with exactly one class definition. The timing parameters in a filter augment data variables in the operation specification, thus realizing data sensitive real-time constraints. A filter also constrains its associated operation, a consequence of the logic defined in Section 2, thus realizing process sensitive real-time constraints.

The advantages of this approach are: (1) the specifications describing functionalities and their refinements remain reusable in contexts where time constraints are not relevant;

consequently, changes to the functional requirements can be analyzed and specified independent from specifying changes to timing constraints; (2) time dependent system properties can be better understood by inspecting only those mappings that relate timing constraints to data and processes of interest; and (3) the timed structure of the system can be used to derive run time scheduling of processes, which will respect time dependent properties and hence safety and liveness properties. A formal verification of real-time systems is possible using the axioms given in Section 2.

Section 2 starts with a bried introduction to Object-Z notation and then describes in detail the notation for RTOZ. An informal semantics and a logic for RTOZ are also given in this section. A case study in robotics has been described in Section 3 along with its functional and timing requirements. We have introduced an object-oriented design of the case study. In Section 4, we give the formal specification for the case study in RTOZ; both functional descriptions and filter specifications for all the components in the case study are given in this section. We give a rigorous proof for the safety and liveness properties of the case study in Section 5. Since the formal verification steps for the case study will be too many to be illustrated concisely, we have given a formal proof of liveness property for a simple example in the Appendix. The paper concludes in Section 6 with a summary and remarks on future work.

2 Specification Notation

This section introduces the formalism that we use to formally specify a real-time system.

2.1 Object-Z Notation

We assume that readers are familiar with Z notaton [15, 17] and Object-Z notation [4]. Only a brief outline of Object-Z notation is given in this section.

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* ($\parallel$), *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 $Op_1 \fatsemi Op_2$ 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.

2.2 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.

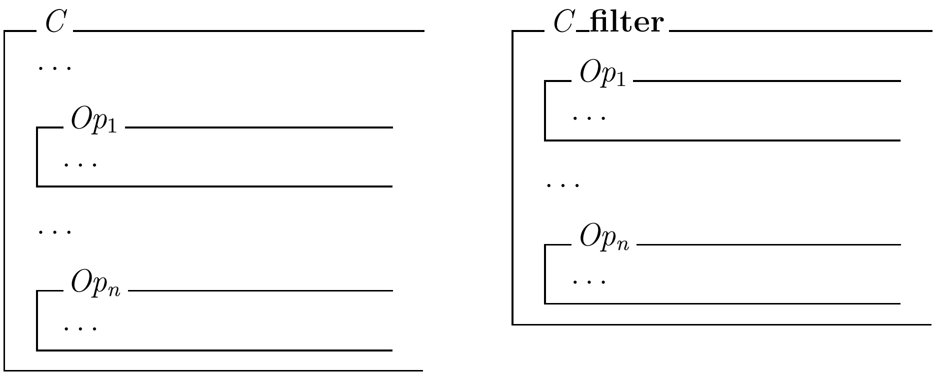


Figure 2: A Class and its Filter Specification

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

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

These two constraints assert that an input can be received at the time of invocation or sometime later, but strictly before the operation terminates. A value can be output strictly sometime after invocation.

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.

2.3 Model of Time in RTOZ

The formal model of time in RTOZ is based on the history of values possessed by data objects[‡] in the application. For a data object x, $History(x)$ denotes a monotonically increasing sequence of real-time values denoting the set of time points at which x has changed in the past. $History(x)$ thus maps the states of x onto a global time scale and projects only those time points at which the value of x has changed; other points on the time scale are not important as far as the history of x is concerned. Formally,

$$
\begin{array}{l}
[X] \\
\hline
History : X \rightarrow \mathrm{seq}_0\mathrm{RealTime} \\
\hline
\forall\, x : X \bullet (\forall\, i : 0 \,..\, \#History(x) - 1 \bullet History(x)(i) < History(x)(i+1) \wedge \\
\quad (\forall\, \tau_1 : RealTime \mid History(x)(i) \leq \tau_1 < History(x)(i+1) \bullet \\
\qquad VALUE[x, \tau_1] = VALUE[x, History(x)(i)]))
\end{array}
$$

The notation seq_0 denotes a sequence whose index starts at 0, instead of 1; by default, sequences in Z start at 1. The function $VALUE[x, \tau]$ returns the value of x at time τ, where x is any data object. When a filter is applied to its corresponding operation, the functional variables in the operation are suffixed with the timing variables in the filter. The interpretation of such suffixed variables is formally defined by the function $VALUE[x, \tau]$. The relationship between such a suffixed variable, the *VALUE* function and the history of

[‡]The term 'data object' refers to a variable of simple type such as integer or real, a variable of set type, a variable of sequence type, a variable of schema type, a variable of a Cartesian product type or an object.

the variable is described by the following rule:

$$\frac{History(x)}{x_\tau = VALUE[x, History(x)(\#History(x))]} \quad [\ \tau \geq History(x)(\#History(x))\ ]$$

2.4 Informal Semantics of the Filter Notation

In this section, we describe the informal semantics of the filter notation introduced by us:

- For every class definition, there is exactly one filter specification. If the class name is C, then the filter specification has the name C_**filter**.
- For every operation Op in a class definition C, there is exactly one filter Op (with the same name as that of the operation) in the filter specification C_**filter**. The filter specification C_**filter** contains nothing other than the filters corresponding to the operations of C.
- The timing variables derived from the problem domain are declared globally outside the filter specifications. These global timing variables can be accessed by all filter specifications.
- A filter Op maps timing variables with the functional variables of its corresponding operation Op. The $INIT$ schema is treated similar to operations and hence it has its own filter named $INIT$. Other constructs within a class definition such as *visibility list, type definitions, constant definitions, state space schema* are not affected by the filter.
- The timing variables of a filter Op include those declared within Op and a subset of global timing variables.
- Besides the mapping, a filter can introduce additional constraints between the timing variables.
- A filter cannot introduce new functional variables.
- A filter Op cannot be invoked indendepently; it can only be applied to the operation Op. Similarly, in RTOZ, an operation Op cannot be invoked without applying its filter Op to it.
- If an operation Op is composed from several operations $Op_1, \ldots, Op_n$ using one or more of the composition operators $\wedge$, $\fatsemi$, $\|$, $[]$ and $\bullet$, then the filter Op is defined as a logical conjunction of the filters $Op_1, \ldots, Op_n$. The interpretation here is that the filter Op is composed logically from the filters $Op_1, \ldots, Op_n$.

We now give the semantics for the time dependent behavior of a composition operator and for computing the anticipated completion time of the composite operation. We use the dot notation of Object-Z to denote the invocation time of an operation Op (namely $Op.t$) and the anticipated completion time of Op (namely $Op.\delta t$).

- Let $Op = Op_1 \wedge Op_2$. In this case, both Op_1 and Op_2 proceed independently. Hence both Op_1 and Op_2 are invoked at the same time when Op is invoked and the anticipated completion time of Op is the maximum of the anticipated completion times of the two operations.
- Let $Op = Op_1 \mathbin{⨾} Op_2$. In this case, Op_1 is invoked first, at the time when Op is invoked and Op_2 is invoked after Op_1 is completed. The anticipated completion time of Op is the sum of anticipated completion times of Op_1 and Op_2.
- If $Op = Op_1 [] Op_2$, then only one of the operation will be invoked non-deterministically. Therefore the invocation time of whichever operation is invoked is the same as that of Op (we do not take into account the time taken to decide which operation is to be invoked). The anticipated completion time of Op is the completion time of the operation being invoked.
- The parallel communication operator $\parallel$ in Object-Z does not make the direction of communication explicit. This poses a problem in RTOZ. For example, in the expression $Op_1 \parallel Op_2$, one has to know whether Op_1 (Op_2) sends data to Op_2 (Op_1) or both operations exchange data in a sequential manner. Otherwise, reasoning about their start and completion times is difficult. We resolve this problem in RTOZ by introduing four new parallel operators which make the communication explicit.

  The operator $\overset{\parallel}{\leftarrow}$ in the expression $Op_1 \overset{\parallel}{\leftarrow} Op_2$ indicates unidirectional communication from Op_2 to Op_1. The operator $\overset{\parallel}{\rightarrow}$ in the expression $Op_1 \overset{\parallel}{\rightarrow} Op_2$ indicates unidirectional communication from Op_1 to Op_2. The operator $\overset{\parallel}{\leftrightarrows}$ in the expression $Op_1 \overset{\parallel}{\leftrightarrows} Op_2$ indicates that both Op_1 and Op_2 exchange data but Op_2 initiates the communication. Notice that there may be several data exchanges between Op_1 and Op_2 but we are interested only at the first communication. Using this we can identify which operation has to be scheduled first. The operator $\overset{\parallel}{\rightleftarrows}$ in the expression $Op_1 \overset{\parallel}{\rightleftarrows} Op_2$ similarly indicates bidirectional communication with Op_1 initiating the communication.

  In the expression $Op = Op_1 \theta Op_2$ where θ is one of the parallel communication operators, the invocation time of the operation which initiates the communication is the same as that of Op. The other operation is invoked only after the first operation sends the data. The operation Op is completed only when both the operations terminate. See the logic for Real-Time Object-Z given in the next section for a formal derivation of the invocation times of Op_1 and Op_2 and the anticipated completion time of Op.

  - In the expression $Op = Op_1 \bullet Op_2$, both Op_1 and Op_2 are invoked at the same time when Op is invoked. The anticipated completion time of Op is the maximum of completion times of Op_1 and Op_2.

- When a filter Op is applied to its corresponding operation Op, the result is a timed specification of the operation Op in which the functional variables are augmented with timing variables according to the filter specification. This is illustrated by the following example:
Operation:

$$
\begin{array}{|l}
\hline \textit{Op} \\
\Delta(x, y) \\
p? : \mathbb{N} \\
\hline
x < p? \Rightarrow y' = p? \\
x \geq p? \Rightarrow y' = y \\
\hline
\end{array}
$$

Filter:

$$
\begin{array}{|l}
\hline \textit{Op} \\
p? : \mathbb{N} \\
t : \textit{RealTime} \\
\delta t, \epsilon_{1p?} : \textit{Time} \\
\hline
\textit{mapping} = \{x \mapsto t, y \mapsto t, p? \mapsto t + \epsilon_{1p?}, y' \mapsto t + \delta t\} \\
\hline
\end{array}
$$

Result of Applying Filter to Operation:

$$
\begin{array}{|l}
\hline \textit{Op} \\
\Delta(x, y) \\
p? : \mathbb{N} \\
t : \textit{RealTime} \\
\delta t, \epsilon_{1p?} : \textit{Time} \\
\hline
x_t < p?_{t+\epsilon_{1p?}} \Rightarrow y'_{t+\delta t} = p?_{t+\epsilon_{1p?}} \\
x_t \geq p?_{t+\epsilon_{1p?}} \Rightarrow y'_{t+\delta t} = y_t \\
\hline
\end{array}
$$

The semantics of the notation v_t can be described as follows:

  - If v is a simple variable (of type $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$), then v_t denotes the value of v at time t.

- If v is an object (of type C, where C is a class definition), then unless otherwise specified, every attribute of C (defined in the state space of C) will be augmented with t and the semantics is applied to each attribute separately. Since each attribute of C can be individually identified, it is also possible to augment different timing variables for different attributes of the same class. The following example illustrates this concept: Consider a part description in a mechanical parts assembly system.

  Part

  $id : \mathbb{N}$
  $status : PartStatus$
  $position : Point$

  When a part obj is initialized at time t_0, it has a precise position and status and hence it is justified to write obj_{t_0} to denote $obj.id_{t_0}$, $obj.status_{t_0}$ and $obj.position_{t_0}$. If the part is assembled or removed from the environment at a later time t ($t > t_0$), its status and position will change at time t but its id will not change. So, we may want the id of the object at time t to be the same as the id of the object at time t_0 but its status and position are updated at time t. This can be stated as

  $$obj.id'_{t_1} = obj.id_{t_0} \land obj.status'_{t_1} = obj.status_t \land obj.position'_{t_1} = obj.position_t$$

- If v is a set (of type power set, denoted by $\mathbb{P}$), then every element of the set is augmented with the same time t. The interpretation is that all the set elements are observed at the same time t, no matter at what time the individual elements are created; i.e., $(s : \mathbb{P}\,X;\ x : X \mid x \in s) \vdash s_t \Rightarrow x_t \in s_t$
  This can be easily justified from the fact that x is known only as an element of the set s. Hence, for any time t_1, the expression $x_{t_1},\ t_1 < t \lor t_1 > t$ is meaningless since x_{t_1} is not guaranteed to be a member of the set s at time t_1.

- If v is a tuple (of type Cartesian product), then unless otherwise specified all elements of the tuple are augmented with the same time t. For example, if $v = (p, q)$ then $v_t = (p_t,\ q_t)$. It is also possible to assign different timing variables to the components of a tuple as in $v_t = (p_{t_1}, q_{t_2})$ where $t_1 \leq t, t_2 \leq t$.

- If v is a sequence (denoted by seq or iseq), then unless otherwise stated, every element of the sequence is augmented with the same time t. Since it is possible to individually select an element from a sequence by subscripting, one can augment different timing variables for different elements of the sequence. The following two examples illustrate this concept:
  Example 1:

  $$v : \operatorname{iseq} \mathbb{N} \Rightarrow$$
  $$(v_t \Rightarrow \forall\, i : 1 \,..\, \#v \bullet v(i)_t$$

Example 2:

$v : \mathrm{iseq}\,\mathbb{N};\ x, y, z : \mathbb{N} \Rightarrow$

$v_t = \langle x_{t_1}, y_{t_2}, z_{t_3} \rangle \wedge t_1 \leq t, t_2 \leq t, t_3 \leq t$

- If v is a function name (the function being defined using axiomatic declaration within the class definition), then no timing variable is associated with v. This is justified based on the fact that Object-Z (and Z) treats axiomatic descriptions as constants. However, parameters of v in this case, will be augmented with timing variables.

- There exists a fixed time t_0 of type *RealTime* which denotes the initialization time of the system. Every variable t of type *RealTime* used in the specification will be greater than or equal to t_0. All the variables of the filter *INIT* in the filter specification corresponding to the root class (the class that describes the system as a whole or that describes the environment) will be augmented with t_0.

2.4.1 Inheritance

Inheritance in Object-Z is defined as a mechanism for incremental specification. That is, a subclass C_{sub} can include additional features after inheriting from a superclass C_{sup}. Even if an inherited operation Op_{sup} is redefined in the subclass as Op_{sub} (keeping the same name), the result of inheritance is a conjunction of the two operations; i.e., $Op \mathrel{\widehat=} Op_{sub} \wedge Op_{sup}$. Object-Z imposes the restriction that Op_{sub} cannot contradict Op_{sup}; should this happen, one of the operations has to be renamed. This justifies the fact that inheritance in Object-Z is an incremental mechanism.

We extend the inheritance mechanism to the filter specifications. Thus, if a class C_{sub} inherits a class C_{sup}, then the filter specification C_{sub}**_filter** inherits the filter specification C_{sup}**_filter**. There is one difference: If a filter Op is redefined in C_{sub}, the timing predicate of $C_{sub}.Op$ is conjoined with that of $C_{sup}.Op$; the **mapping** of $C_{sub}.Op$ overrides that of $C_{sup}.Op$. In other words, there will be only one consistent **mapping** in the result after inheritance. The syntax for inherting a filter specification is the same as inheriting a class into another; i.e.,

C_{sub}**_filter**

C_{sup}**_filter**

...

and the semantics is

C_{sup}**_filter** $\subset$ C_{sub}**_filter**

We insist on having a different name for the filter specification corresponding to the subclass because of the semantics of the filter notations; i.e., there should be a one-to-one correspondence between class definitions and filter specifications. Notice that the relationship between the filter specifications is $\subset$ because the subclass has to be necessarily different from the superclass. This necessitates the filter specification corresponding to the subclass to have additional filters. Consequently, every new filter in C_{sub}**_filter** will be due to one of the following reasons:

- A state variable v in C_{sup} is renamed as w in C_{sub}. In this case, every occurrence of v in C_{sup}**_filter** has to be renamed by w in C_{sub}**_filter**. That is,
  $C_{sub}\_\mathbf{filter}.Op \mathrel{\widehat=} C_{sup}\_\mathbf{filter}.Op[w/v]$

- An operation Op_1 in C_{sup} is renamed as Op_2 in C_{sub}. In this case, a new filter has to be defined for Op_2 in C_{sub}**_filter** which will include the same mapping as that of Op_1 in C_{sup}**_filter**. That is,
  $C_{sub}\_\mathbf{filter}.Op_2 \mathrel{\widehat=} C_{sup}\_\mathbf{filter}.Op_1 \wedge \neg\ \exists\, C_{sub}\_\mathbf{filter}.Op_1$

- An additional state variable v is introduced in C_{sub}. Let Op_1, Op_2, ..., Op_n, $n \geq 0$ be the operations in C_{sup} which are modified due to the addition of v. Since inheritance is only incremental, these operations will have additional constraints due to the addition of v. Consequently, there will be new operation definitions Op_1, Op_2, ..., Op_n in C_{sub}, where each Op_i, $1 \leq i \leq n$ will include additional constraints. There will be new filters corresponding to $Op_i, 1 \leq i \leq n$ in C_{sub}**_filter**. Each of these filters will include additional mapping corresponding to the additional constraints in their respective functional definitions. These filters may also have additional timing requirements. Formally,
  $C_{sub}\_\mathbf{filter}.Op_i \supset C_{sup}\_\mathbf{filter}.Op_i,\ 1 \leq i \leq n$

- An additional operation Op_{new} is introduced in C_{sub}. Consequently, there will be a filter Op_{new} in C_{sub}**_filter**.

- A combination of all of the above situations is also possible.

2.4.2 Polymorphism

Object-Z permits universal polymorphism whereby an object of a subclass C_{sub} can be substituted for an object of its superclass C_{sup}. The syntax for polymorphic references in Object-Z is

$c : \downarrow C_{sup}$

The additional methods defined in C_{sub} cannot be invoked by c. When c invokes a method Op, two cases arise:

- If Op is defined in C_{sup} and is not redefined in C_{sub}, then Op in C_{sup} will be enabled.

- If Op is defined in C_{sup} and is redefined in C_{sub}, then the logical conjunction of $C_{sup}.Op$ and $C_{sub}.Op$ will be enabled (according to the semantics of inheritance).

Case 1 will not pose any problems in our approach. However, case 2 deserves special attention. The condition for universal polymorphism is that if an object O_{sub} belonging to C_{sub}

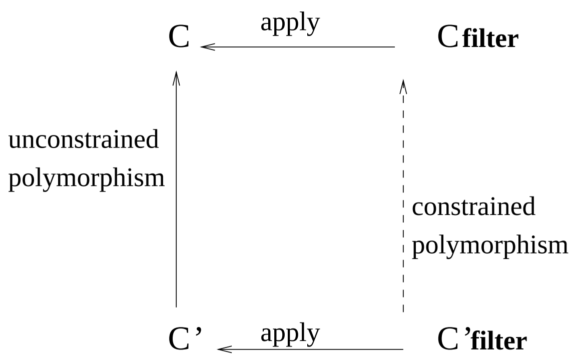


Figure 3: Timing restrictions under polymorphic substitutions

is substituted for an object O_{sup} belonging to C_{sup}, then O_{sub} should behave identically to that of O_{sup} in all respects. That is, the filter specification C_{sub}**_filter** should be more restrictive that the filter specification C_{sup}**_filter**. In other words, the timing requirements of any operation invoked by O_{sub} must satisfy the timing requirements expected or defined within the filter specification for O_{sup}. Since filters cannot be invoked independently, this restriction can be realized only after applying the filters to their corresponding operations. Figure 3 shows the relationship between unconstrained (without timing restrictions) polymorphism and constrained polymorphism (with timing restrictions).

The timing requirements in a filter consists of two parts - mapping between the functional variables and timing variables, and additional constraints on timing variables. Since Op in C_{sub} can only define additional constraints (incremental specification by inheritance) which should be consistent with the set of constraints already defined in Op in C_{sup}, the timing requirements in filter C_{sub}**_filter**.Op can have additional maplets and/or additional timing constraints. Using this fact, we give the semantics for polymorphic substitution as follows:

- For every filter Op in C_{sub}**_filter**, the mapping of Op should be a superset of the mapping of filter Op in C_{sup}**_filter** and the timing constraints in the former should imply the timing constraint in the latter.

2.5 Logic for Real-Time Object-Z

This section presents a logic for Real-Time Object-Z. The logic for Object-Z [13] extends the logic for Z[] in a conservative manner. The logic described in this section builds on top of this conservative extension to provide an abstract, axiomatic semantics for time dependent data and processes. For the purposes of this paper, we use the term "data object" to refer to a variable of simple type such as integer or real, a variable of set type, a variable of sequence type, a variable of schema type, a variable of Cartesian product type or an object. In all these cases, a data object possesses a value which may be simple or a collection of values. The notion of "data object" enables us to report its state and history.

We define a new data type called seq_0 which is a sequence starting with an index 0, instead of 1. Formally,

$$\mathrm{dom}\,\mathrm{seq}_0[\mathrm{X}] == \mathrm{dom}\,\mathrm{seq}\,\mathrm{X} \cup \{0\}$$

2.5.1 Meta-functions used in the logic for Object-Z

The logic for Object-Z [13] uses several operators and meta-functions for simplifying the presentation of rules. Below, we have listed those operators and meta-functions which we have used in this paper.

Operators:

- $\odot$ denotes the substitution of a binding into an expression, predicate or declaration.
- $\circledast$ denotes the substitution of types, attributes and primed variables of an object into a predicate.

Meta-functions:

- ϕ returns the free variables of a declaration or a schema or an expression.
- α returns the set of name bindings of a data object.
- α_1 and α_2 respectively return the names of primary and secondary variables of a class.
- Ω returns the set of names of all operations in a class definition.
- $\beta_?$ and $\beta_!$ return the basenames of inputs and outputs of an operation respectively.
- δ returns the set of names of the variables in the delta list of an operation of a class.
- ι returns the set of inherited classes (the class definitions) of a class.

2.5.2 Additional Meta-functions

We define the following additional meta-functions to simplify the real-time expressions in the proof rules:

- ω returns the name of the operation or the name of the filter, passed as parameter. Since a filter for an operation has the same name as that of the operation, we use the same meta_function to extract the name of the operation as well as the name of its filter.
- Π_c returns the name of a class; the function Π_{str} returns the name of a filter specification. Actually, Π_{str} strips off the keyword **_filter** in the name of a filter specification and returns only the user defined name of the filter specification. The purpose here is to compare it with that of its corresponding class's name.

- α_T returns the set of names of timing variables involved in a filter.
- Υ returns the mapping of functional variables to timing variables in a filter.
- Γ_t returns the timing predicate of a filter.
- $\mathcal{D}$ returns the declaration of an operation or a filter. It includes the delta variables and the input and output parameters that are explicitly declared in this operation/filter and those that are inherited.
- $\mathcal{P}$ returns the predicate of an operation or a filter. It is the conjunction of the explicitly declared predicate and those of the inherited operations/filters.
- $VALUE[x, \tau]$ returns the value of x at time τ, where x is any data object.
- *last* accepts a data object x and a time τ to return a time τ_1 such that $\tau_1 \leq \tau$ and the duration from τ_1 to τ is the longest duration during which $VALUE[x, \tau_1]$ is held.

2.5.3 Inference Rules

Following the notations in [13, 17], we give inference rules. A rule has the following format:

Rule: (*Rule name*)

$$\frac{\textit{Premises}}{\textit{Conclusion}} \quad [\ \textit{Proviso}\ ]$$

The premises are given as a list of sequents:

$$\textit{Premises} \mathrel{\widehat{=}} \textit{Sequent} \ldots \textit{Sequent}$$

and the conclusion is given as a single sequent:

$$\textit{Conclusion} \mathrel{\widehat{=}} \textit{Sequent}$$

The term *Proviso* is a predicate in conjunctive normal form. In writing the *Proviso*, line breaks between two predicates indicate an implicit conjunction.

A sequent represents a step in the derivation process. Each sequent accepts a set of specifications and asserts a condition (expressed by a predicate). Formally, a sequent is

$$d \mid \Psi \vdash \Phi$$

where d is a list of declarations, Ψ is a set of predicates and Φ is another set of predicates. The term $d \mid \Psi$ denotes the *antecedent* of the sequent and Φ denotes the *consequent*.

Variables We first define history associated with each data object and then formalize the notion of the value of a variable at time τ. For a data object x, $History(x)$ is a monotonically increasing sequence of real-time values denoting the set of time points at which x has changed in the past. $History(x)$ thus maps the states of x onto a global time scale and projects only those time points at which the value of x has changed; other points on the time scale are not important as far as the history of x is concerned. Formally,

$$
\begin{array}{l}
[X] \\
\hline
History : X \rightarrow \mathrm{seq_0 RealTime} \\
\hline
\forall\, x : X \bullet \\
\quad (\forall\, i : 0 \mathinner{..} \#History(x) - 1 \bullet \\
\qquad History(x)(i) < History(x)(i+1) \wedge \\
\qquad (\forall\, \tau_1 : RealTime \mid History(x)(i) \leq \tau_1 < History(x)(i+1) \bullet \\
\qquad\quad VALUE[x, \tau_1] = VALUE[x, History(x)(i)]))
\end{array}
$$

We next establish the relationship between the function *last* and *History*.

$$last(x, \tau) = History(x)(i), \qquad History(x)(i) \leq \tau < History(x)(i+1)$$

The inference rule "CurrentValue" defined below asserts x_τ the value of x at time τ. This value is the most recently held value of x on or before τ.

<u>Rule:</u> (CurrentValue)

$$\frac{History(x) \vdash}{\vdash x_\tau = VALUE[x, History(x)(\#History(x))]} \quad [\ \tau \geq History(x)(\#History(x))\ ]$$

The operators $+$ and $-$ are coercive with respective to the two types *RealTime* and *Time*; the resulting expression in each case is of type *RealTime*.

<u>Rule:</u> ($\tau \pm \delta\tau$)

$$\frac{\tau : RealTime \quad \delta\tau : Time \vdash}{\tau \pm \delta\tau : RealTime \vdash}$$

Expressions The application of "mapping" defined in a filter OP_2 to its corresponding operation OP_1 is defined by the following rule: The functional variables $fnvars_2$ declared in OP_2 must be a subset of the functional variables $fnvars_1$ involved in OP_1. The application of "mapping" produces a new predicate part for OP_1 in which each functional variable fnv in $fnvars_1 \cap fnvars_2$ is replaced by fnv_τ, where the maplet $fnv \mapsto \tau$ is found in the predicate part of OP_2.

In order to formally define this rule, we have to first define the substitution for expressions and then promote this substitution to predicates. We use the notation $M \leadsto E$ to denote the substitution of the mapping M to the expression E. Table 1 a list of rules governing the substitution of M to E.

Semantics	Description
$M \leadsto (x) = \begin{cases} x_\tau & \textbf{if } (x \mapsto \tau) \in M \\ x & \textit{otherwise} \end{cases}$	Application to a variable.
$M \leadsto x[e] = \begin{cases} x_\tau[M \leadsto e] & \textbf{if } (x \mapsto \tau) \in M \\ x[M \leadsto e] & \textit{otherwise} \end{cases}$	Application to a generic expression.
$M \leadsto (i) = i$	Application to a numeral.
$M \leadsto \{e_1, \ldots, e_n\} = \{M \leadsto e_1, \ldots, M \leadsto e_n\}$	Application to set enumeration.
$M \leadsto \{S \bullet e\} = \{M \leadsto S \bullet M \leadsto e\}$	Application to set comprehension.
$M \leadsto (\mathbb{P}\, e) = \mathbb{P}(M \leadsto e)$	Application to set declaration.
$M \leadsto (e_1, \ldots, e_n) = (M \leadsto e_1, \ldots, M \leadsto e_n)$	Application to a tuple.
$M \leadsto (s_1 \times \ldots \times s_n) = (s_1 \times \ldots \times s_n)$	Application to a Cartesian product.
$M \leadsto (e.i) = (M \leadsto e).i$	Application to tuple selection.
$M \leadsto \langle\!\vert\, x_1 := e_1, \ldots, x_n := e_n \,\vert\!\rangle =$ $\langle\!\vert\, x_1 := M \leadsto e_1, \ldots, x_n := M \leadsto e_n \,\vert\!\rangle,$ $x_i \notin \mathrm{dom}\, M, 1 \leq i \leq n$	Application to a binding.
$M \leadsto (\theta S) = \theta S$	Application to theta expression.
$M \leadsto (b.x) = (M \leadsto b).x$	Application to binding selection.
$M \leadsto (f\ e) = f(M \leadsto e)$	Application to a function application.
$M \leadsto (\mu\, S \bullet e) = (\mu\, S \bullet (M \leadsto e))$	Application to a μ expression.
$M(\ \textbf{if}\ P\ \textbf{then}\ e_1\ \textbf{else}\ e_2) =$ $\textbf{if}\ (M \leadsto P)\ \textbf{then}\ (M \leadsto e_1)\ \textbf{else}\ (M \leadsto e_2)$	Application to an **if-then-else** expression.
$M \leadsto (b \odot e) = (M \leadsto b) \odot (M \leadsto e)$	Application to binding application.

Table 1: Substitution of time-variables mapping into expressions

Predicates We use the same notation $\leadsto$ for describing the substitution M in a predicate. Rules in table 2 describe the substitutions for predicates.
The proof rule asserting the relationship between the four timing parameters in a filter is given below:

Rule: (Timing Parameters)

$$\frac{OP = \{d \mid p\}}{\vdash \delta\tau > 0 \wedge (0 \leq \epsilon_{1x} < \delta\tau) \wedge (0 < \epsilon_{2y} \leq \delta\tau)} \quad [\, p_1 \,]$$

where $p \equiv x \in \beta_?(OP) \wedge y \in \beta_!(OP)$
$(\tau : \mathit{RealTime}) \in d \wedge (\delta\tau, \epsilon_{1x}, \epsilon_{2y} : \mathit{Time}) \in d$

Semantics	Description
$M \rightsquigarrow (e \in s) = (M \rightsquigarrow e) \in (M \rightsquigarrow s)$	Application to set membership.
$M \rightsquigarrow (e = v) = (M \rightsquigarrow e) = (M \rightsquigarrow v)$	Application to equality.
$M \rightsquigarrow (\neg P) = \neg (M \rightsquigarrow P)$	Application to negation.
$M \rightsquigarrow (P \land Q) = (M \rightsquigarrow P) \land (M \rightsquigarrow Q)$	Application to conjunction.
$M \rightsquigarrow (P \lor Q) = (M \rightsquigarrow P) \lor (M \rightsquigarrow Q)$	Application to disjunction.
$M \rightsquigarrow (P \Rightarrow Q) = (M \rightsquigarrow P) \Rightarrow (M \rightsquigarrow Q)$	Application to implication.
$M \rightsquigarrow (P \Leftrightarrow Q) = (M \rightsquigarrow P) \Leftrightarrow (M \rightsquigarrow Q)$	Application to equivalence.
$M \rightsquigarrow (\forall S \bullet P) = \forall (M \rightsquigarrow S) \bullet (M \rightsquigarrow P)$	Application to universally quantified expressions.
$M \rightsquigarrow (\exists S \bullet P) = \exists (M \rightsquigarrow S) \bullet (M \rightsquigarrow P)$	
$M \rightsquigarrow (\exists_1 S \bullet P) = \exists_1 (M \rightsquigarrow S) \bullet (M \rightsquigarrow P)$	Application to existentially quantified expressions.
$M \rightsquigarrow (S) = (M \rightsquigarrow S)$	Application to parenthesised expressions.
$M \rightsquigarrow (b \odot P) = (M \rightsquigarrow b) \odot (M \rightsquigarrow P)$	Application to a binding and a predicate.

Table 2: Substiution of time-variables mapping into predicates

Applying Filters The necessary and sufficient condition for applying a filter OP_2 to an operation OP_1 are: (i) both OP_1 and OP_2 have the same name; and (ii) the functional variables in the declaration of OP_2 must be a subset of the functional variables in the declaration of OP_1. This second condition enforces that a filter cannot introduce new functional variables. The result of applying OP_2 to OP_1 will be a new operation OP which is formed as follows: (i) the declaration part of OP is the union of declarations of OP_1 and OP_2; (ii) the predicate part of OP is a conjunction of the predicate part of OP_1 substituted by "mapping" of OP_2, and the timing predicate of OP_2. Filter application is denoted by $\hookrightarrow$. Formally,

<u>Rule:</u> (ApplyFilter $\hookrightarrow$)

$$\frac{OP_1 = \{d_1 \mid p_1\} \vdash \qquad OP_2 = \{d_2 \mid p_2\} \vdash}{OP_2 \hookrightarrow OP_1 \vdash OP = \{d \mid p\}} \quad [\, p \,]$$

where $p \equiv \omega(OP) = \omega(OP_1) = \omega(OP_2)$
$\qquad \mathcal{D}(OP_2) \setminus \alpha_T(OP_2) \subseteq \mathcal{D}(OP_1)$
$\qquad d = d_1 \cup d_2$
$\qquad p = (\Upsilon(OP_2) \rightsquigarrow p_1) \land \Gamma_t(OP_2)$

Composite Filters In this section, we give the proof rules for applying a filter OP when its associated operation OP is defined using one or more of the composition operators - $\land$ (conjunction operator), $\fatsemi$ (sequential composition operator), $\parallel$ (parallel composition operator), $[]$ (choice operator) and $\bullet$ (environment enrichment operator).

If an operation OP is defined as $OP = OP_1 \theta OP_2$ where θ is a composition operator, the filter OP must use the filters OP_1 and OP_2. We do not, however, compose the filter OP with the same composition operator θ. This makes sense because a filter cannot be invoked independently. One has to apply the filter to its associated operation first. For this reason, we always conjoin the filters to form a composite filter.

The proof rules below mandate how the invocation time t of OP is to be propagated to the invocation times of OP_1 and OP_2 and how the anticipated completion time of OP is obtained from the completion times of OP_1 and OP_2. We use the dot notation (e.g., $OP.t$) in the proof rules to refer to the individual timing variables in a filter OP.

Conjunction Let $OP = OP_1 \wedge OP_2$. Both OP_1 and OP_2 proceed independent of each other. Accordingly, the invocation times of OP_1 and OP_2 are the same as that of OP. The completion time of OP is the maximum of the completion times of OP_1 and OP_2.

<u>Rule:</u> (Conjunction)

$$\frac{OP = OP_1 \wedge OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = OP.t \wedge \\ OP.\delta t = max(OP_1.\delta t, OP_2.\delta t)\end{array}}$$

Sequential Composition Let $OP = OP_1 \mathbin{;} OP_2$. OP_2 will be invoked only after OP_1 is completed. Accordingly, the invocation time of OP_1 is the same as that of OP and the invocation time of OP_2 is $t + OP_1.\delta t$. The completion time of OP is the sum of the completion times of OP_1 and OP_2.

<u>Rule:</u> (Sequential Composition)

$$\frac{OP = OP_1 \mathbin{;} OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = t + OP_1.\delta t \wedge \\ OP.\delta t = OP_1.\delta t + OP_2.\delta t\end{array}}$$

Parallel Operator In Object-Z, the expression $OP_1 \parallel OP_2$ denotes the communication between OP_1 and OP_2 through data exchange. The communication may be unidirectional or bidirectional. However, Object-Z semantics does not make precise the direction of communication. This poses a problem in RTOZ when OP_1 and OP_2 are composed using $\parallel$ operator. We resolve this problem by explicitly identifying the direction of communication.

The expression $OP_1 \overset{\parallel}{\leftarrow} OP_2$ denotes that OP_1 receives data from OP_2; the expression $OP_1 \overset{\parallel}{\rightarrow} OP_2$ denotes that OP_2 receives data from OP_1; the expression $OP_1 \overset{\parallel}{\leftrightarrows} OP_2$ denotes that both OP_1 and OP_2 exchange data with each other, but OP_2 sends the output first before it receives an input from OP_1; and the expression $OP_1 \overset{\parallel}{\rightleftarrows} OP_2$ denotes that

OP_1 sends an output first before it receives an input from OP_2. Notice that there may be any number of data exchanges between OP_1 and OP_2, but the notation makes it clear as to which operation acts first. Proof rules for the four expressions follow:

<u>Rule:</u> (Parallel Left)

$$\frac{OP = OP_1 \overset{\parallel}{\leftarrow} OP_2 \vdash}{\begin{array}{l}\vdash OP_2.t = OP.t \wedge OP_1.t = OP.t + OP_2.\epsilon_{2x} \wedge \\ OP.\delta t = max(OP_2.\delta t, OP_1.\delta t + OP_2.\epsilon_{2x})\end{array}} \quad [\ x \in \beta_?(OP_1) \wedge x \in \beta_!(OP_2)\ ]$$

<u>Rule:</u> (Parallel Right)

$$\frac{OP = OP_1 \overset{\parallel}{\rightarrow} OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = OP.t + OP_1.\epsilon_{2x} \wedge \\ OP.\delta t = max(OP_1.\delta t, OP_2.\delta t + OP_1.\epsilon_{2x})\end{array}} \quad [\ x \in \beta_?(OP_2) \wedge x \in \beta_!(OP_1)\ ]$$

<u>Rule:</u> (Parallel LeftRight)

$$\frac{OP = OP_1 \overset{\parallel}{\leftrightarrows} OP_2 \vdash}{\begin{array}{l}\vdash OP_2.t = OP.t \wedge OP_1.t = OP.t + OP_2.\epsilon_{2x} \wedge \\ Op_2.t + OP_2.\epsilon_{1x} \geq OP_1.t + OP_1.\epsilon_{2x} \wedge \\ OP.\delta t = max(OP_2.\delta t, OP_1.\delta t + OP_2.\epsilon_{2x})\end{array}} \quad [\ x \in \beta_?(OP_1) \wedge x \in \beta_!(OP_2)\ ]$$

<u>Rule:</u> (Parallel RightLeft)

$$\frac{OP = OP_1 \overset{\parallel}{\rightleftarrows} OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = OP.t + OP_1.\epsilon_{2x} \wedge \\ OP_1.t + OP_1.\epsilon_{1x} = OP_2.t + OP_2.\epsilon_{2x} \wedge \\ OP.\delta t = max(OP_1.\delta t, OP_2.\delta t + OP_1.\epsilon_{2x})\end{array}} \quad [\ x \in \beta_?(OP_2) \wedge x \in \beta_!(OP_1)\ ]$$

Choice Operator If $OP = OP_1[]OP_2$, either OP_1 or OP_2, but not both, will be invoked depending on which pre-condition is satisfied. Whichever operation is selected for invocation, its invocation time is the same as that of OP. The completion time of OP is the same as that of the invoked operation.

<u>Rule:</u> (Choice)

$$\frac{OP = OP_1[]OP_2 \vdash}{\begin{array}{l}\vdash (OP_1.t = OP.t \wedge OP.\delta t = OP_1.\delta t) \bigoplus \\ (OP_2.t = OP.t \wedge OP.\delta t = OP_2.\delta t)\end{array}}$$

The notation $\bigoplus$ denotes *exclusive-OR*.

Environment Enrichment Let $OP = OP_1 \bullet OP_2$. Both OP_1 and OP_2 are merged together to get OP. This is the same as schema conjunction in Z. Therefore, the invocation times of OP_1 and OP_2 are the same as that of OP; the completion time of OP is the maximum of completion times of OP_1 and OP_2.

Rule: (Enrichment)

$$\frac{OP = OP_1 \bullet OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = OP.t \wedge \\ OP.\delta t = max(OP_1.\delta t, OP_2.\delta t)\end{array}}$$

Implication and Equivalence In addition to the operators defined above, an operation schema in Object-Z can also be specified using the logical connectives *conjunction* ($\wedge$), *disjunction* ($\vee$), *negation* ($\neg$), *implication* ($\Rightarrow$) and *equivalence* ($\Leftrightarrow$). Among these, logical conjunction is the same as the conjunction operator discussed earlier. We now give proof rules for operations defined using the other four connectives; these rules are self explanatory.

Rule: (Disjunction)

$$\frac{OP = OP_1 \vee OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = OP.t \wedge \\ OP.\delta t = max(OP_1.\delta t, OP_2.\delta t)\end{array}}$$

Rule: (Negation)

$$\frac{OP = \neg\, OP_1 \vdash}{\vdash OP_1.t = OP.t \wedge OP.\delta t = OP_1.\delta t}$$

Rule: (Implication)

$$\frac{OP = OP_1 \Rightarrow OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = OP.t \wedge \\ OP.\delta t = \textbf{ if } \mathcal{P}(OP_1) \textbf{ then } OP_1.\delta t + OP_2.\delta t \textbf{ else } OP_1.\delta t\end{array}}$$

Rule: (Equivalence)

$$\frac{OP = OP_1 \Leftrightarrow OP_2 \vdash}{\begin{array}{l}\vdash OP_1.t = OP.t \wedge OP_2.t = OP.t \wedge \\ OP.\delta t = max(OP_1.\delta t, OP_2.\delta t)\end{array}}$$

The completion time of the composite operation OP in the expression $OP = OP_1 \vee OP_2$ is the maximum of completion times of OP_1 and OP_2 since both operations can be invoked simultaneously. We have ignored the time to check for the logical disjunction. Other rules can be interpreted in a similar way.

Binding a class and its filter specification Next, we give the proof rule for binding a filter specification $C\_\mathbf{filter}$ to its corresponding class definition C. The result is a timed class definition C_r in which the filters of $C\_\mathbf{filter}$ are applied to their corresponding operations in C. We introduce the notation $\rightrightarrows$ to denote the binding of $C\_\mathbf{filter}$ to C. A formal definition of this rule follows:

Rule: (ClassFilterBinding $\rightrightarrows$)

$$\frac{C ::\mid OP_1 \in \Omega(C) \vdash \qquad C\_\mathbf{filter} ::\mid OP_2 \in \Omega(C\_\mathbf{filter}) \vdash}{C_r ::= (C\_\mathbf{filter} :: \rightrightarrows\ C ::) \vdash OP \in \Omega(C_r)} \quad [\, p \,]$$

where $p \equiv \Pi_c(C_r) = \Pi_c(C) = \Pi_{str}(C\_\mathbf{filter})$
$\Omega(C_r) = \Omega(C) = \Omega(C\_\mathbf{filter})$
$OP = OP_2 \hookrightarrow OP_1$
$C_r.INIT = C\_\mathbf{filter}.INIT \hookrightarrow C.INIT$

The rule 'ClassFilterBinding' asserts that OP_1 must be an operation from class C, OP_2 is its filter derived from the filter specification $C\_\mathbf{filter}$ and OP is the timed operation derived by applying OP_2 to OP_1. The $INIT$ filter of $C\_\mathbf{filter}$ is also applied in a similar way to the $INIT$ schema of C.

Invoking an operation In RTOZ, a filter cannot be invoked independently without the operation to which it applies. Moreover, an operation can be invoked only after its filter is applied to it. We specify this condition by the following rule:

Rule: (OperationInvocation)

$$\frac{\vdash a \in C \quad C ::\mid OP_1 \in \Omega(C) \vdash \quad C\_\mathbf{filter} ::\mid OP_2 \in \Omega(C\_\mathbf{filter}) \vdash}{\begin{array}{l} \vdash a.OP = \big[\, a \circledast \mathcal{D}(OP_1) \cup a \circledast \mathcal{D}(OP_2) \mid \\ \qquad a \circledast (\mathcal{P}(OP_2) \rightsquigarrow \mathcal{P}(OP_1) \wedge \\ \qquad\quad x_1' = x_1 \wedge \ldots x_n' = x_n) \big] \\ \qquad\quad \setminus (y_1, \ldots, y_m, y_1', \ldots, y_m') \end{array}} \quad [\, p \,]$$

where $p \equiv \omega(OP) = \omega(OP_1) = \omega(OP_2)$
$\alpha_1(C) \cup \alpha_1(C\_\mathbf{filter}) \setminus \delta\ (C, OP_1) = \{x_1, \ldots, x_n\}$
$\alpha_1(C) \cup \alpha_1(C\_\mathbf{filter}) \cup \alpha_2(C) \cup \alpha_2(C\_\mathbf{filter}) = \{y_1, \ldots, y_m\}$
$\phi(OP_1) \subseteq \alpha(C)$
$\phi(OP_2) \subseteq \alpha(C\_\mathbf{filter})$

Inheritance When a filter specification $C\_\mathbf{filter}_1$ inherits another filter specification $C\_\mathbf{filter}_2$, the inference rules for class inheritance as given in [13] are directly applicable. This is due to the fact a filter specification is an Object-Z specification. However, the rule for inheriting operations are to be redefined. This is because when a filter OP_1 in $C\_\mathbf{filter}_1$ is inherited into a filter OP_2 in $C\_\mathbf{filter}_2$, the timing predicate of OP_1 and that of OP_2 are conjoined; the mapping of OP_2 overrides the mapping of OP_1. In other words, there will be only one consistent "mapping" in the result after inheritance.

In specifying this rule, we closely follow the convention in [13]. The premises include the set of filter specifications $C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_n$ which are inherited. The sequence $C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_n$ denotes the inheritance chain with $C\_\mathbf{filter}_n$ representing the most recent subclass inheriting all other filter specifications. Thus, the inheriting filter OP will be derived from each of these filter specifications. A formal definition of the rule for inheritance follows:

<u>Rule:</u> (FilterInheritance)

$$\frac{C\_\mathbf{filter}_1 ::| \; OP_1 \in \Omega(C\_\mathbf{filter}_1) \vdash \ldots \; C\_\mathbf{filter}_n ::| \; OP_n \in \Omega(C\_\mathbf{filter}_n) \vdash}{\begin{array}{l} C\_\mathbf{filter} ::| \; OP \in \Omega(C\_\mathbf{filter}) \vdash\uparrow OP = \big[\mathcal{D}(OP_1) \cup \ldots \cup \mathcal{D}(OP_n) \mid \\ \quad \Upsilon(OP) = \Upsilon(OP_n) \oplus \ldots \oplus \Upsilon(OP_1) \wedge \\ \quad \Gamma_t(OP_1) \wedge \ldots \wedge \Gamma_t(OP_n)\big] \end{array}} \quad [\, p \,]$$

$$\begin{array}{ll} \text{where } p \equiv & \omega(\uparrow OP) = \omega(OP_1) = \ldots = \omega(OP_n) \\ & \iota(C\_\mathbf{filter}) = \{C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_n\} \\ & \phi(OP_1) \subseteq \alpha(C\_\mathbf{filter}_1) \\ & \vdots \\ & \phi(OP_n) \subseteq \alpha(C\_\mathbf{filter}_n) \end{array}$$

Polymorphic Substitution In the proof rule for polymorphic substitutions, we assert that an object a is polymorphic when it belongs to one of the subclasses of a given class, provided that the filter specification is already applied to the subclass.

<u>Rule:</u> (Polymorphism)

$$\frac{\vdash a \in\downarrow (C\_\mathbf{filter} \;\rightrightarrows\; C)}{\begin{array}{l} \vdash a \in (C\_\mathbf{filter}_1 \;\rightrightarrows\; C_1) \bigoplus \ldots \bigoplus \\ \quad a \in (C\_\mathbf{filter}_n \;\rightrightarrows\; C_n) \end{array}} \quad [\, p \,]$$

$$\begin{array}{ll} \text{where } p \equiv & (\iota^{\sim})^{*}(\!|\; C \;|\!) = \{C_1, \ldots, C_n\} \\ & (\iota^{\sim})^{*}(\!|\; C\_\mathbf{filter} \;|\!) = \{C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_n\} \end{array}$$

The expression $(\iota^{\sim})^{*}(\!|\; C \;|\!) = \{C_1, \ldots, C_n\}$ denotes that the classes $C_1, \ldots, C_n$ are descendants (subclasses) of the class C. In the conclusion of the rule for polymorphic substitution, we assert that object a belongs to only one of the descendent classes of C. This is consistent with the OO definitions for *Class* and *Object* that we follow, namely an object belongs to only one class.

3 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 4.

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 3.3) 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 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 [8] 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 towards formal verification.

3.1 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

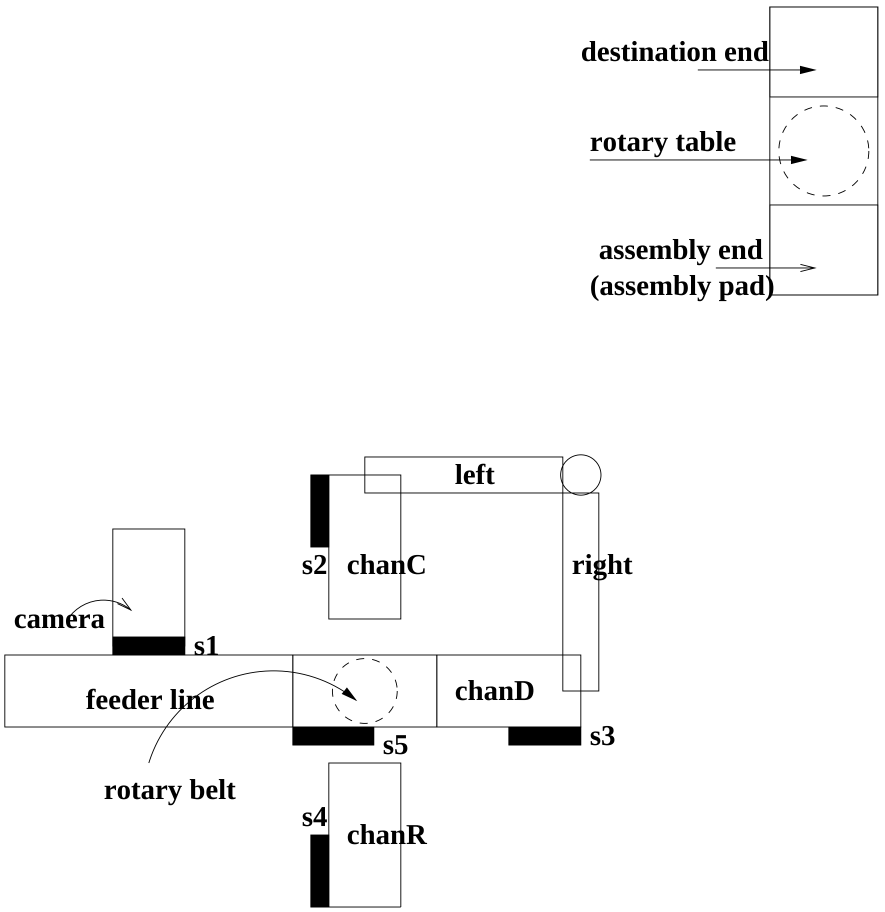


Figure 4: Top View of the Assembly Cell

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.

3.2 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.

3.2.1 Assumptions on Environment

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

3.2.2 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).

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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 [11].

3.3 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.
  $t_{fb} < t_{ro} + 2 * t_d$

3.4 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.

3.5 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 5 and 6 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.

4 Formal Specification of the Case Study

In this section, we give the behavioral specification of two subsystems modeling the case study. 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.

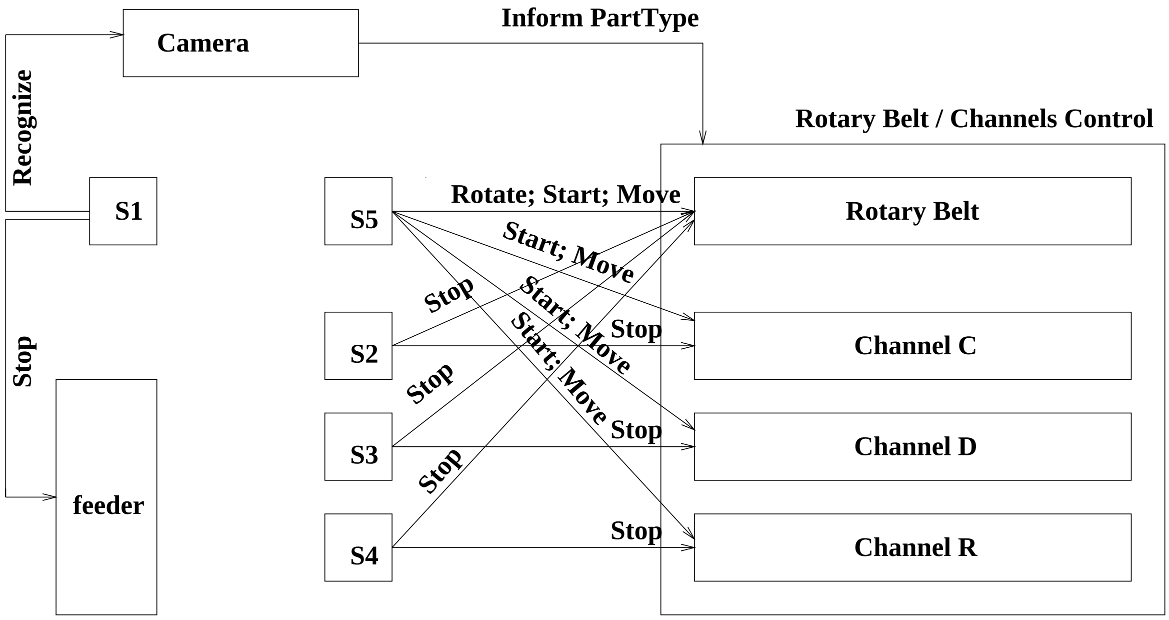


Figure 5: Architecture of the Parts Recognition Subsystem

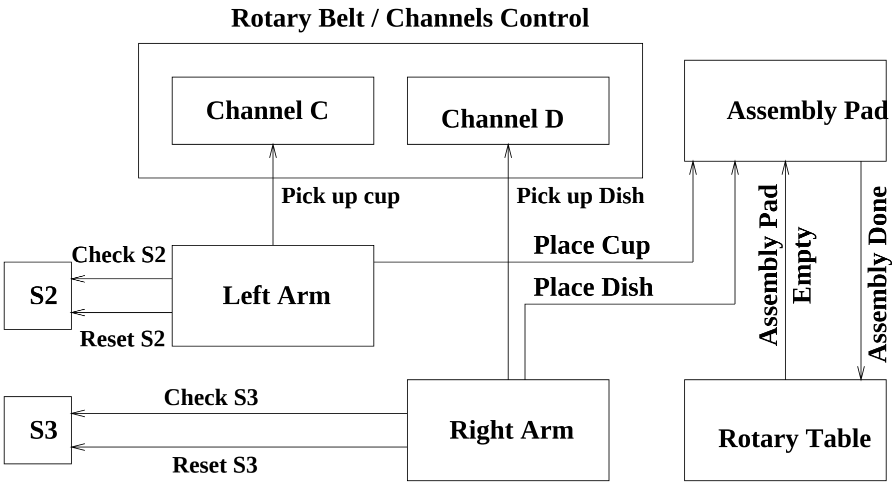


Figure 6: Architecture of the Assembly Subsystem

4.1 Object-Z Specification

A formal behavioral specification of the whole system can be obtained by composing individual class definitions of basic components. We first give the global types and variables definition and then informally describe the functionalities of the classes used by other specifications in this paper.

Type definitions and constant definitions needed by all classes are shown in Figure 7.

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

$$
\begin{array}{|l}
constant\_distance : \mathbb{R} \\
camera\_position : Point \\
collector\_position : Point \\
assembly\_pad\_position : Point \\
leftarm\_initial\_position : Point \\
rightarm\_initial\_position : Point
\end{array}
$$

Figure 7: Type Definitions and Constant Definitions

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.

4.1.1 Specifying Classes

For each component in the OO model shown in Figures 5 and 6, we give a class specification in this section.

Part

$id : \mathbb{N}$
$position : Point$
$status : PartStatus$
$type : PartType$

ChangePosition
$\Delta(position)$
$new? : Point$

$position' = new?$

ChangeStatus
$\Delta(status)$
$newst? : PartStatus$

$status' = newst?$

Figure 8: Specification of the class *Part*

The specification in Figure 8 introduces the features and operations on parts. A part has a position, status, a unique i.d. and a type. There are two operations defined for a part - to change its position and to change its status. The i.d. and type of a part cannot be changed. Since the visibility list is empty, all features of *Part* are exported.

Camera

$status : CameraStatus$

INIT
$status = IDLE$

Recognize
$\Delta(status)$
$part? : Part$
$type! : PartType$

$part?.position = camera\_position$
$status = IDLE$
$status' = RECOGNIZE$
$type! = part?.type$

Idle
$\Delta(status)$

$status = RECOGNIZE$
$status' = IDLE$

Figure 9: Specification for the class *Camera*

A camera has *status* as its only attribute and it is used to indicate whether the camera is currently recognizing an object or is idle. Initially, it is idle. The position of the camera is fixed and is known to the environment. The specification for the class *Camera* shown in Figure 9 has three operations describing the functionality of a camera. When the camera recognizes an object, its status changes from *IDLE* to *RECOGNIZE*. After the operation, the type of the object in front of the camera is returned as the result, which is available to any other object invoking *Recognize* operation. Since visibility list is empty, all features of *Camera* are exported.

Belt

$$\mathsf{IsOnBelt\_} : \mathbb{P}\, Point$$

$$\begin{array}{l}
start_of_belt : Point \\
end_of_belt : Point \\
objects : \mathrm{iseq}\, Part \\
status : BeltStatus \\
\hline
\forall\, i, j : 1 \mathinner{..} \#objects \bullet \\
\quad (i \neq j \Rightarrow \\
\qquad object(i).id \neq objects(j).id \land \\
\qquad object(i).position \neq objects(j).position) \land \\
\qquad objects(i).status = ON_BELT \land \\
\qquad \mathsf{IsOnBelt}(objects(i).position) \\
\mathsf{IsOnBelt}(start_of_belt) \land \mathsf{IsOnBelt}(end_of_belt) \\
start_of_belt \neq end_of_belt
\end{array}$$

INIT

$$\begin{array}{l}
objects = \langle\rangle \\
status = STOPPED
\end{array}$$

Start

$$\begin{array}{l}
\Delta(status) \\
\hline
status = STOPPED \\
status' = MOVING
\end{array}$$

Stop

$$\begin{array}{l}
\Delta(status) \\
\hline
status = MOVING \\
status' = STOPPED
\end{array}$$

Move

$$\begin{array}{l}
\Delta(objects) \\
dist? : \mathbb{R} \\
out! : \mathrm{iseq}\, Part \\
\hline
status = MOVING \\
objects = objects' \frown out! \\
\forall\, i : 1 \mathinner{..} \#objects' \bullet (\exists_1\, j : 1 \mathinner{..} \#objects \mid objects'(i).id = objects(j).id \bullet \\
\quad objects'(i).position = objects(j).position + dist? \land \\
\quad \mathsf{IsOnBelt}(objects'(i).position)) \\
\forall\, obj : Part \mid obj \in \mathrm{ran}\, out! \bullet \\
\quad \neg\, (\mathsf{IsOnBelt}(obj.position) \lor obj.position = end_of_belt
\end{array}$$

Figure 10: Specification for the class *Belt*

$$
\begin{array}{|l}
\hline
\textit{Belt(Contd...)} \\
\begin{array}{|l}
\hline
\textit{Place} \\
\Delta(objects) \\
newobj? : Part \\
\hline
newobj?.id \notin \{obj : Part \mid obj \in objects \bullet obj.id\} \\
\quad objects' = \langle(\mu\, obj : Part \mid obj.id = newobj?.id \land obj.status = ON\_BELT \land \\
\qquad\qquad obj.position = start\_of\_belt)\rangle \frown objects \\
\hline
\end{array} \\
\hline
\end{array}
$$

Figure 11: Specification for the class *Belt* (Contd...)

The class *Belt* in Figures 10 and 11 specifies the general characteristics of all the belts. A belt is modeled using the set of parts on the belt. Every belt is of finite length and has distinct start and end points. The status of the belt indicates whether it is moving or is stopped; initially, all the belts are stopped. The state invariant for a belt ensures that (i) no two parts on the belt occupy the same position, (ii) the status of all parts on the belt is *ON_BELT* and (iii) the position of every part on the belt is within the belt region. We have used a function IsOnBelt to ensure the last condition. However we have not elaborated this function in the specification because the type *Point* is assumed to be a basic type.

There are four operations defined for a belt - *Start, Stop, Move* and *Place*. The operations *Start* and *Stop* denote the status when the belt is started and is stopped. *Move* operation specifies the constraints to move a belt through a distance *dist*?. It asserts that the status of the belt should be *MOVING* when this operation is invoked. The *Move* operation collectively moves all parts on the belt. Due to movement, some parts may go out of the belt (indicated by *out*! in *Move*) and others will stay (indicated by *objs* in *Move*). Finally, the *Place* operation defines the constraints for placing a part on the belt; it asserts that the i.d. of the new part should not be the same as that of any other part on the belt and the new part is placed at the start of the belt.

$$
\begin{array}{l}
\textit{RotaryBeltChannels} \\
\hline
\quad rotary\_belt, chanC, chanD, chanR : Belt \\
\quad parttype : PartType \\
\quad tablepos : RotaryBeltPosition \\
\hline
\quad \{obj : Part \mid obj \in \mathrm{ran}(rotary\_belt.objects) \bullet obj.id\} \cap \\
\quad \{obj : Part \mid obj \in \mathrm{ran}(chanC.objects) \bullet obj.id\} \cap \\
\quad \{obj : Part \mid obj \in \mathrm{ran}(chanD.objects) \bullet obj.id\} \cap \\
\quad \{obj : Part \mid obj \in \mathrm{ran}(chanR.objects) \bullet obj.id\} = \varnothing \\
\\
\quad \textit{INIT} \\
\hline
\quad rotary\_belt.INIT \\
\quad chanC.INIT \\
\quad chanD.INIT \\
\quad chanR.INIT \\
\quad tablepos = ZERO \\
\quad rotary\_belt.end\_of\_belt = chanD.start\_of\_belt \\
\\
\quad \textit{RotateNinety} \\
\hline
\quad \Delta(tablepos, rotary\_belt) \\
\quad dir? : Direction \\
\hline
\quad (tablepos = ZERO \wedge dir? = ANTI\_CLOCKWISE) \Rightarrow \\
\qquad (tablepos' = NINETY \wedge \\
\qquad\; rotary\_belt.end\_of\_belt' = chanC.start\_of\_belt) \\
\quad (tablepos = ZERO \wedge dir? = CLOCKWISE) \Rightarrow \\
\qquad (tablepos' = TWO\_SEVENTY \wedge \\
\qquad\; rotary\_belt.end\_of\_belt' = chanR.start\_of\_belt) \\
\quad (tablepos = NINETY \wedge dir? = CLOCKWISE) \Rightarrow \\
\qquad (tablepos' = ZERO \wedge \\
\qquad\; rotary\_belt.end\_of\_belt = chanD.start\_of\_belt) \\
\quad (tablepos = TWO\_SEVENTY \wedge dir? = ANTI\_CLOCKWISE) \Rightarrow \\
\qquad (tablepos' = ZERO \wedge \\
\qquad\; rotary\_belt.end\_of\_belt = chanD.start\_of\_belt)
\end{array}
$$

Figure 12: Specification for the class *RotaryBeltChannels*

$RotaryBeltChannels(Contd...)$

$Start\_Belts\_RBC$

$\Delta(rotary\_belt, chanC)$

$rotary\_belt.status = STOPPED \wedge chanC.status = STOPPED$
$rotary\_belt.status' = MOVING \wedge chanC.status' = MOVING$

$Start\_Belts\_RBD$

$\Delta(rotary\_belt, chanD)$

$rotary\_belt.status = STOPPED \wedge chanD.status = STOPPED$
$rotary\_belt.status' = MOVING \wedge chanD.status' = MOVING$

$Start\_Belts\_RBR$

$\Delta(rotary\_belt, chanR)$

$rotary\_belt.status = STOPPED \wedge chanR.status = STOPPED$
$rotary\_belt.status' = MOVING \wedge chanR.status' = MOVING$

$Stop\_Belts\_RBC$

$\Delta(rotary\_belt, chanC)$

$rotary\_belt.status = MOVING \wedge chanC.status = MOVING$
$rotary\_belt.status' = STOPPED \wedge chanC.status' = STOPPED$

$Stop\_Belts\_RBD$

$\Delta(rotary\_belt, chanD)$

$rotary\_belt.status = MOVING \wedge chanD.status = MOVING$
$rotary\_belt.status' = STOPPED \wedge chanD.status' = STOPPED$

$Stop\_Belts\_RBR$

$\Delta(rotary\_belt, chanR)$

$rotary\_belt.status = MOVING \wedge chanR.status = MOVING$
$rotary\_belt.status' = STOPPED \wedge chanR.status' = STOPPED$

Figure 13: Specification for the class *RotaryBeltChannels* (Contd...)

$$\textit{RotaryBeltChannels}(\textit{Contd...})$$

$$
\begin{aligned}
&Move\_Belts\_RBC \mathrel{\widehat=} (rotary\_belt.Move[constant\_distance/dist?] \bullet \\
&\qquad [\, out!(\#out!).position = rotary\_belt.end\_of\_belt \,]) \fatsemi \\
&\qquad chanC.Place[out!(\#out!)/part?]) \fatsemi chanC.Move[constant\_distance/dist?] \\
&Move\_Belts\_RBD \mathrel{\widehat=} (rotary\_belt.Move[constant\_distance/dist?] \bullet \\
&\qquad [\, out!(\#out!).position = rotary\_belt.end\_of\_belt \,]) \fatsemi \\
&\qquad chanD.Place[out!(\#out!)/part?] \fatsemi chanD.Move[constant\_distance/dist?] \\
&Move\_Belts\_RBR \mathrel{\widehat=} (rotary\_belt.Move[constant\_distance/dist?] \bullet \\
&\qquad [\, out!(\#out!).position = rotary\_belt.end\_of\_belt \,]) \fatsemi \\
&\qquad chanR.Place[out!(\#out!)/part?] \fatsemi chanR.Move[constant\_distance/dist?]
\end{aligned}
$$

Figure 14: Specification for the class *RotaryBeltChannels* (Contd...)

Figures 12 through 14 show the specification for the class *RotaryBeltChannels*. It models the control mechanism for the rotary belt and the three channel belts. Since the channel belts are controlled by the same mechanism as that of the rotary belt, creating an additional aggregate class such as *RotaryBeltChannels* is appropriate.

The variable *tablepos* in this class indicates the current position of the rotary belt. Initially it is in position ZERO and the rotary belt is aligned with the feeder line and channel D (refer to Figure 1). The position $NINETY$ refers to the situation when the rotary belt is rotated counter clockwise from its initial position so that the tail end of the rotary belt is aligned with channel C. Similarly, the position $TWO\_SEVENTY$ refers to the situation when the rotary belt is rotated clockwise and is aligned with the return channel. The operation *RotateNinety* enumerates all possible rotations including the reset of the rotary belt.

The class *RotaryBeltChannels* also includes operations by which the rotary belt can be started and stopped simultaneously with the channel belts. This is necessary when a part at the front end of the rotary belt has to be fed to the appropriate channel. In addition, the class also includes three move operations corresponding to the three channels. All the three move operations are identical in their behavior. The operation *Move_Belts_RDC* describes that (i) the part at the front end of the rotary belt is moved to the tail of the rotary belt; (ii) the same part is placed at the front end of the channel C and (iii) it is moved to the tail end of the channel C. All features of the class *RotaryBeltChannels* are exported.

Sensor

$id : \mathbb{N}_1$
$position : Point$
$status : SensorStatus$

$\textsc{Init}$

$status = OFF$

ChangeStatus

$\Delta(status)$

$status = ON \Rightarrow status' = OFF$
$status = OFF \Rightarrow status' = ON$

Figure 15: Specification for the class *Sensor*

The *Sensor* class shown in Figure 15 models a sensor. Each sensor is characterized by a unique i.d., the position of the sensor and its status (initially OFF). There is only one operation that can be performed on a sensor - toggle its status. All features of *Sensor* are visible.

The *RobotArm* class shown in Figure 16 models a robot arm, which will later be instantiated to create the left and right arms of the robot. The robot arm moves in a predefined trajectory. The position of the arm is a point on this trajectory. The status of a robot arm indicates whether or not it is currently holding a part. Initially, the status is *EMPTY*. There is an internal operation to change the status of the robot arm. There are two exported operations - *Pick* and *Place*. The operation *Pick* describes the situation when a robot arms picks up an object from one of the channels. The pre-condition for this operation asserts that the arm is empty and the part is in a different position. The post-condition asserts that the part has been grasped; i.e., position of the part is the same as position of the arm and status of the part is changed to *GRASPED*. The robot arm also changes its status from *EMPTY* to *HOLDING*. The operation *Place* specifies in a similar way the palcement of a part onto the assembly pad.

As seen from the class specification of *RobotArm*, the internal structure of the robot, namely its position and status, is hidden from the environment; only the operations *Pick* and *Place* are exported.

$$
\begin{array}{l}
\textit{RobotArm} \\
\upharpoonright(\textit{Pick}, \textit{Place}) \\
\hline
\textit{position} : \textit{Point} \\
\textit{status} : \textit{ArmStatus} \\
\hline
\textit{INIT} \\
\textit{status} = \textit{EMPTY} \\
\hline
\textit{ChangeStatus} \\
\Delta(\textit{status}) \\
\hline
\textit{status}' = \ \mathbf{if} \ \textit{status} = \textit{HOLDING} \ \mathbf{then} \ \textit{EMPTY} \\
\qquad \mathbf{else} \ \ \mathbf{if} \ \textit{status} = \textit{EMPTY} \ \mathbf{then} \ \textit{HOLDING} \\
\hline
\textit{Pick} \mathrel{\widehat=} [\, \textit{status} = \textit{EMPTY} \,] \wedge \\
\quad [\, \textit{obj}? : \textit{Part} \mid \textit{obj}?.\textit{position} \neq \textit{position} \wedge \textit{obj}?.\textit{status} = \textit{ON\_BELT} \,] \bullet \\
\quad \textit{ChangeStatus} \wedge \textit{obj}?.\textit{ChangePosition}[\textit{position}/\textit{new}?] \wedge \\
\quad [\, \textit{newst}? : \textit{PartStatus} \mid \textit{newst}? = \textit{GRASPED} \,] \bullet \\
\qquad \textit{obj}?.\textit{ChangeStatus} \\
\textit{Place} \mathrel{\widehat=} [\, \textit{status} = \textit{HOLDING} \,] \wedge \\
\quad [\, \textit{obj}? : \textit{Part} \mid \textit{obj}.\textit{position} = \textit{position} \wedge \textit{obj}?.\textit{status} = \textit{GRASPED} \,] \bullet \\
\quad \textit{ChangeStatus} \wedge \textit{obj}?.\textit{ChangePosition}[\textit{assembly\_pad\_position}/\textit{new}?] \wedge \\
\quad [\, \textit{newst}? : \textit{PartStatus} \mid \textit{newst}? = \textit{ON\_ASSEMBLY\_PAD} \,] \bullet \\
\qquad \textit{obj}?.\textit{ChangeStatus}
\end{array}
$$

Figure 16: Specification for the class *RobotArm*

The class *AssembledPart* shown in Figure 17 represents the assembly of a cup and dish pair. As asserted by its state invariant, the assembly has a unique i.d. which is different from that of the cup and that of the dish. By virtue of the assembly process, the position of the assembly is the same as the position of the cup and the position of the dish. *ChangePosition*, which changes the position of an assembled pair is the only operation defined in this class. All features from this class are exported.

$$
\begin{array}{|l}
\hline
\textit{AssembledPart} \\
\hline
\quad \begin{array}{|l}
\hline
id : \mathbb{N}_1 \\
position : Point \\
cup, dish : Part \\
\hline
cup.parttype = CUP \land dish.parttype = DISH \\
cup.position = dish.position = position \\
cup.status = ASSEMBLED \land dish.status = ASSEMBLED \\
\hline
\end{array} \\
\\
\quad \begin{array}{|l}
\hline
\textit{ChangePosition} \\
\hline
\Delta(position) \\
new? : Point \\
\hline
position' = new? \\
\hline
\end{array} \\
\hline
\end{array}
$$

Figure 17: Specification for the class *AssembledPart*

A specification for the class *AssemblyPad* is shown in Figure 18. It describes the assembly pad where the actual assembly is performed. The pad has a fixed position which is known to the environment. The status of the assembly pad indicates whether the pad is currently empty, or has a dish or a cup and dish pair. Initially, the assembly pad is empty and so its status is $FREE$.

There is only one operation defined in this class - *PerformAssembly*. It accepts a cup and a dish as inputs. These two are placed on the assembly pad before performing the assembly and hence the position of the cup and the position of the dish are the same which are also equal to the position of the assembly pad. As a result of this operation, a new assembly is created from the cup and dish. The result is conveyed through a variable set to $TRUE$.

AssemblyPad

$$
\begin{array}{l}
position : Point \\
status : Assembly\_Pad\_Status \\
\hline
position = assembly\_pad\_position
\end{array}
$$

INIT

$$
status = FREE
$$

PerformAssembly

$$
\begin{array}{l}
cup?, dish? : Part \\
assembly! : AssembledPart \\
result! : Boolean \\
\hline
status = PAIR\_IS\_THERE \\
cup?.position = dish?.position = position \\
assembly!.cup = cup? \land assembly!.dish = dish? \\
result! = TRUE
\end{array}
$$

Figure 18: Specification for the class *AssemblyPad*

The *RotaryTable* class specification shown in Figure 19 describes a rotary table, with one of its ends acting as the assembly pad. This is asserted by the state invariant in this class. It has a boolean variable *done* which is initially set to false. This variable will be set to true when a pair of cup and dish are assembled at the assembly end. Typically, it is the communication variable between *AssemblyPad* and this class. The operation *Rotate* asserts that after an assembly, the table is rotated so that the assembled pair is transferred to the destination end of the table.

RotaryTable

$assembly\_end, destination\_end : Point$
$done : Boolean$

$assembly\_end \neq destination\_end$

INIT

$assembly\_end = assembly\_pad\_position$
$done = FALSE$

Rotate

$\Delta(assembly\_end, destination\_end, done)$
$original? : AssembledPart$

$original?.position = assembly\_end$
$done = TRUE$
$assembly\_end' = destination\_end$
$destination\_end' = assembly\_end$
$done' = FALSE$

Figure 19: Specification for the class *RotaryTable*

4.1.2 Specifying Subsystems

Having defined the individual classes, we now give the specification for the two subsystems - *PartsRecognitionSubsystem* and *AssemblySubsystem*, obtained by composing the classes specified in earlier sections.

PartsRecognitionSubsystem

$$
\begin{array}{l}
feeder : Belt \\
rbc : RotaryBeltChannels \\
camera : Camera \\
s1, s2, s3, s4, s5 : Sensor \\
\hline
distance(feeder.start\_of\_belt, camera.position) = \\
\quad distance(feeder.end\_of\_belt, camera.position) = constant\_distance \\
feeder.\ \mathsf{IsOnBelt}\ (camera.position) \\
(\forall\, i, j : 1 \mathinner{.\,.} \#(feeder.objects) \mid abs(i - j) = 1 \bullet \\
\quad distance((feeder.objects(i)).position, (feeder.objects(j)).position) \\
\qquad = constance\_distance) \\
\{ obj : Part \mid obj \in \operatorname{ran} feeder.objects \bullet obj.id \} \cap \\
\quad \{ obj : Part \mid obj \in (\operatorname{ran} rbc.rotary\_belt.objects \cup \\
\qquad \operatorname{ran} rbc.chanC.objects \cup \operatorname{ran} rbc.chanD.objects \cup \\
\qquad \operatorname{ran} rbc.chanR.objects) \bullet obj.id \} = \varnothing \\
rbc.rotary\_belt.tablepos = ZERO \Rightarrow \\
\quad feeder.end\_of\_belt = rbc.rotary\_belt.start\_of\_belt \\
s1.id \neq s2.id \neq s3.id \neq s4.id \neq s5.id \\
s1.position = camera.position
\end{array}
$$

INIT

$$
\begin{array}{l}
feeder.INIT \\
rbc.INIT \\
camera.INIT \\
s1.INIT \land s2.INIT \land s3.INIT \land s4.INIT \land s5.INIT
\end{array}
$$

$$
\begin{array}{l}
AddPart \mathrel{\widehat=} [\, newpart? : Part \,] \bullet \\
\quad \neg\ [\, obj : Part \mid obj \in \operatorname{ran} feeder.objects \land \\
\qquad distance(obj.position, feeder.start\_of\_belt) < constant\_distance \,] \land \\
\quad feeder.Place[newpart?/newobj?]
\end{array}
$$

$$
\begin{array}{l}
RecognizePart \mathrel{\widehat=} [\, \exists\, obj : Part \bullet obj \in \operatorname{ran} feeder.objects \land \\
\quad obj.position = camera.position \,] \bullet \\
\qquad s1.ChangeStatus \land feeder.Stop \land \\
\qquad (camera.Recognize[obj/part?] \bullet [\, rbc.parttype = type! \,])
\end{array}
$$

Figure 20: Specification for Parts Recognition Subsystem

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

Figure 21: Specification for Parts Recognition Subsystem (Contd...)

The specification for *PartsRecognitionSubsystem* is shown in Figures 20 and 21. 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.

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.

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

Figure 22: Specification for the class *AssemblySubsystem*

The *AssemblySubsystem* specification is shown in Figure 22. 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.

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*.

4.2 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 logic for Real-Time Object-Z. 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.

4.2.1 Global Timing Variables

Timing variables and constants derived from the problem description are shown in Figure 23; these are declared as global constants so that all filter specifications can refer to them. The constraints given in Section 3.3 are also captured in the filter specifications.

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

Figure 23: Timing Variables and Constants

4.2.2 Filter Specifications and Filters

The filter specifications corresponding to individual classes are given first. This is followed by the filter specifications corresponding to subsystems.

Filter Specifications for Classes

$Part\_\textbf{filter}$

$ChangePosition$

$position, position' : Point$
$new? : Point$
$t : RealTime$
$\delta t, \epsilon_{1new?} : Time$

$mapping = \{position \mapsto t, position' \mapsto t + \delta t, new? \mapsto t + \epsilon_{1new?}\}$

$ChangeStatus$

$status, status' : PartStatus$
$newst? : PartStatus$
$t : RealTime$
$\delta t, \epsilon_{1newst?} : Time$

$mapping = \{status \mapsto t, status' \mapsto t + \delta t, newst? \mapsto t + \epsilon_{1newst?}\}$

Figure 24: Filter Specification corresponding to the class *Part*

The filter specification *Part*_**filter** shown in Figure 24 specifies two filters, corresponding to the two operations *ChangePosition* and *ChangeStatus*. The filter *ChangePosition* includes two state variables *position* and *position′* which are derived from the statement $\Delta(position)$ in the operation *ChangePosition*. The input variable *new*? is also declared in the operation *ChangePosition*. In addition to these functional variables, the filter includes three timing variables: t denotes the invocation time of the filter *ChangePosition* and its associated operation *ChangePosition*, δt denotes the anticipated completion time of the operation *ChangePosition* and $\epsilon_{1new?}$ denotes the offset after the invocation at which the input *new*? is provided to the operation. According to the last predicate given in the filter, the input can be provided any time immediately after the invocation but before the operation is completed. The predicate "$mapping = \ldots$" specifies that *position* must be known at the time of invocation, the input *new*? must be supplied at time $t + \epsilon_{1new?}$ and the value of *position′* in the predicate part of the operation *ChangePosition* is true only after the operation is completed.

The other filter in *Part*_**filter** and all other filter specifications in this chapter can be interpreted in a similar way. When it is necessary, we will provide additional explanations.

$Camera_$filter

INIT

$status : CameraStatus$
$t : RealTime$

$mapping = \{status \mapsto t\}$

Recognize

$status, status' : CameraStatus$
$part? : Part$
$type! : PartType$
$t : RealTime$
$\delta t, \epsilon_{1part?}, \epsilon_{2type!} : Time$

$mapping = \{status \mapsto t, status' \mapsto t + \delta t, part? \mapsto t + \epsilon_{1part?}, type! \mapsto t + \epsilon_{2type!}\}$
$t_r > 0 \wedge t_r \leq \epsilon_{2type!} \leq \delta t$

Idle

$status, status' : CameraStatus$
$t : RealTime$
$\delta t : Time$

$mapping = \{status \mapsto t, status' \mapsto t + \delta t\}$

Figure 25: Filter Specification corresponding to the class *Camera*

The filter *Recognize* shown in Figure 25 asserts that the recognition process takes at least t_r time units as specified by the predicate $t_r \leq \epsilon_{2type!} \leq \delta t$. The same predicate conjoined with the maplet $type! \mapsto t + \epsilon_{2type!}$ asserts that the operation *Recognize* outputs the type of the part being recognized only after the camera recognizes the part. The predicate $0 \leq \epsilon_{1part?} < \delta t$ conjoined with the maplet $part? \mapsto t + \epsilon_{1part?}$ asserts that the input to the operation *Recognize* is given $\epsilon_{1part?}$ units after it is being invoked.

Belt

INIT

$$
\begin{array}{l}
objects : \mathrm{iseq}\, Part \\
status : BeltStatus \\
t : RealTime \\
\hline
mapping = \{objects \mapsto t, status \mapsto t\}
\end{array}
$$

Start

$$
\begin{array}{l}
status, status' : BeltStatus \\
t : RealTime \\
\delta t : Time \\
\hline
mapping = \{status \mapsto t, status' \mapsto t + \delta t\}
\end{array}
$$

Stop

$$
\begin{array}{l}
status, status' : BeltStatus \\
t : RealTime \\
\delta t : Time \\
\hline
mapping = \{status \mapsto t, status' \mapsto t + \delta t\}
\end{array}
$$

Move

$$
\begin{array}{l}
objects, objects' : \mathrm{iseq}\, Part \\
status : BeltStatus \\
dist? : \mathbb{R} \\
out! : \mathrm{iseq}\, Part \\
t : RealTime \\
\delta t, \epsilon_{1 dist?}, \epsilon_{2 out!} : Time \\
\hline
mapping = \{objects \mapsto t, objects' \mapsto t + \delta t, status \mapsto t, \\
\quad dist? \mapsto t + \epsilon_{1 dist?}, out! \mapsto t + \epsilon_{2 out!}\} \\
\delta t \geq dist?/s
\end{array}
$$

Figure 26: Filter Specification corresponding to *Belt*

$Belt\_\mathbf{filter}(Contd...)$

$Place$

$objects, objects' : \mathrm{iseq}\, Part$
$newobj? : Part$
$t : RealTime$
$\delta t, \epsilon_{1newobj?} : Time$

$mapping = \{objects \mapsto t, objects' \mapsto t + \delta t, newobj? \mapsto t + \epsilon_{1newobj?}\}$

Figure 27: Filter Specification corresponding to *Belt* (Contd...)

The description of *Belt*_**filter** as shown in Figures 26 and 27 is self-explanatory. Notice that the filter *Move* asserts that the anticipated completion time is the time taken for the belt to move through the distance *dist*?.

$$
\begin{array}{l}
\textbf{RotaryBeltChannels\_filter} \\
\quad \textit{INIT} \\
\quad rotary\_belt, chanC, chanD, chanR : Belt \\
\quad tablepos : RotaryBeltPosition \\
\quad t : RealTime \\
\quad \hline \\
\quad mapping = \{rotary\_belt \mapsto t, chanC \mapsto t, chanD \mapsto t, chanR \mapsto t, tablepos \mapsto t\} \\
\quad rotary\_belt.INIT[t/rotary\_belt.INIT.t] \\
\quad chanC.INIT[t/chanC.INIT.t] \\
\quad chanD.INIT[t/chanD.INIT.t] \\
\quad chanR.INIT[t/chanR.INIT.t] \\
\\
\quad RotateNinety \\
\quad rotary\_belt, rotary\_belt' : Belt \\
\quad chanC, chanD, chanR : Belt \\
\quad tablepos, tablepos' : RotaryBeltPosition \\
\quad dir? : Direction \\
\quad t : RealTime \\
\quad \delta t, \epsilon_{1dir?} : Time \\
\quad \hline \\
\quad mapping = \{rotary\_belt \mapsto t, rotary\_belt' \mapsto t + \delta t, \\
\qquad chanC \mapsto t, chanD \mapsto t, chanR \mapsto t, \\
\qquad tablepos \mapsto t, tablepos' \mapsto t + \delta t, dir? \mapsto t + \epsilon_{1dir?}\} \\
\quad \delta t \geq t_{ro}
\end{array}
$$

Figure 28: Filter Specification corresponding to *RotaryBeltChannels*

*RotaryBeltChannels*_**filter**(*Contd...*)

Start_Belts_RBC

$$rotary\_belt, rotary\_belt', chanC, chanC' : Belt$$
$$t : RealTime$$
$$\delta t : Time$$

$$mapping = \{rotary\_belt \mapsto t, rotary\_belt' \mapsto t + \delta t, chanC \mapsto t, chanC' \mapsto t + \delta t\}$$

Start_Belts_RBD

$$rotary\_belt, rotary\_belt', chanD, chanD' : Belt$$
$$t : RealTime$$
$$\delta t : Time$$

$$mapping = \{rotary\_belt \mapsto t, rotary\_belt' \mapsto t + \delta t, chanD \mapsto t, chanD' \mapsto t + \delta t\}$$

Start_Belts_RBR

$$rotary\_belt, rotary\_belt', chanR, chanR' : Belt$$
$$t : RealTime$$
$$\delta t : Time$$

$$mapping = \{rotary\_belt \mapsto t, rotary\_belt' \mapsto t + \delta t, chanR \mapsto t, chanR' \mapsto t + \delta t\}$$

Stop_Belts_RBC

$$rotary\_belt, rotary\_belt', chanC, chanC' : Belt$$
$$t : RealTime$$
$$\delta t : Time$$

$$mapping = \{rotary\_belt \mapsto t, rotary\_belt' \mapsto t + \delta t, chanC \mapsto t, chanC' \mapsto t + \delta t\}$$

Stop_Belts_RBD

$$rotary\_belt, rotary\_belt', chanD, chanD' : Belt$$
$$t : RealTime$$
$$\delta t : Time$$

$$mapping = \{rotary\_belt \mapsto t, rotary\_belt' \mapsto t + \delta t, chanD \mapsto t, chanD' \mapsto t + \delta t\}$$

Figure 29: Filter Specification corresponding to *RotaryBeltChannels* (Contd...)

$$
\begin{array}{l}
\textit{RotaryBeltChannels}\_\textbf{filter}(\textit{Contd...}) \\
\hline
\quad \textit{Stop\_Belts\_RBR} \\
\quad rotary\_belt, rotary\_belt', chanR, chanR' : Belt \\
\quad t : RealTime \\
\quad \delta t : Time \\
\quad \hline
\quad mapping = \{rotary\_belt \mapsto t, rotary\_belt' \mapsto t + \delta t, \\
\quad \qquad chanR \mapsto t, chanR' \mapsto t + \delta t\} \\
\quad \hline
Move\_Belts\_RBC \mathrel{\widehat=} \left[\, mapping = \{out! \mapsto t + \epsilon_{2out!}, rotary\_belt \mapsto t\} \,\right] \wedge \\
\qquad rotary\_belt.Move[constant\_distance/dist?, t/rotary\_belt.Move.t] \wedge \\
\qquad chanC.Place[out!(\#out!)/part?, t + rotary\_belt.Move.\delta t/chanC.Place.t] \wedge \\
\qquad chanC.Move[constant\_distance/dist?, \\
\qquad\qquad\quad t + rotary\_belt.Move.\delta t + chanC.place.\delta t/chanC.Move.t] \\
Move\_Belts\_RBD \mathrel{\widehat=} \left[\, mapping = \{out! \mapsto t + \epsilon_{2out!}, rotary\_belt \mapsto t\} \,\right] \wedge \\
\qquad rotary\_belt.Move[constant\_distance/dist?, t/rotary\_belt.Move.t] \wedge \\
\qquad chanD.Place[out!(\#out!)/part?, t + rotary\_belt.Move.\delta t/chanD.Place.t] \wedge \\
\qquad chanD.Move[constant\_distance/dist?, \\
\qquad\qquad\quad t + rotary\_belt.Move.\delta t + chanD.place.\delta t/chanD.Move.t] \\
Move\_Belts\_RBR \mathrel{\widehat=} \left[\, mapping = \{out! \mapsto t + \epsilon_{2out!}, rotary\_belt \mapsto t\} \,\right] \wedge \\
\qquad rotary\_belt.Move[constant\_distance/dist?, t/rotary\_belt.Move.t] \wedge \\
\qquad chanR.Place[out!(\#out!)/part?, t + rotary\_belt.Move.\delta t/chanR.Place.t] \wedge \\
\qquad chanR.Move[constant\_distance/dist?, \\
\qquad\qquad\quad t + rotary\_belt.Move.\delta t + chanR.place.\delta t/chanR.Move.t] \\
\hline
\end{array}
$$

Figure 30: Filter Specification corresponding to *RotaryBeltChannels*(Contd...)

In the $INIT$ filter of the filter specification *RotaryBeltChannels*_**filter** shown in Figures 28–30, the initialization time t is passed to the respective initialization of the components; in other words, all components of the class *RotaryBeltChannels* are initialized simultaneously. The notation x/y in Object-Z denotes the substitution of x for y. Notice that in the expression $chanC.INIT[t/chanC.INIT.t]$ the unqualified t refers to the invocation time of $INIT$ in *RotaryBeltChannels*, whereas the t qualified by $chanC.INIT$ refers to the invocation time of $INIT$ in *chanC*. This notation is consistent with feature selection in object-orientation and is followed throughout this paper.

The filter *RotateNinety* includes the constraint that the anticipated completion time must be at least equal to t_{ro} (the time to rotate the rotary belt through 90°).

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. (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. All the move operations in the filter specification *RotaryBeltChannels*_**filter** are defined in this way.

The filter *Move_Belts_RBC* is defined as a sequential composition of several operations from its component classes. The first operation in this composition, namely $rotary\_belt.Move$ is invoked at the same time the operation *Move_Belts_RBC* is invoked. Therefore, we use the filter $rotary\_belt.Move$ in defining the filter *Move_Belts_RBC* by passing the respective time parameters. Sequential composition results in accumulated timings. For example, in the operation *Move_Belts_RBC*, the invocation of $chanC.Place$ is delayed until $rotary\_belt.Move$ is completed. This is indicated by the substitution of $t + rotary\_belt.Move.\delta t$ for $chanC.Place.t$.

Filters for other move operations are defined similarly.

*Sensor*_**filter**

INIT

$status : SensorStatus$
$t : RealTime$

$mapping = \{status \mapsto t\}$

ChangeStatus

$status, status' : SensorStatus$
$t : RealTime$
$\delta t : Time$

$mapping = \{status \mapsto t, status' \mapsto t + \delta t\}$

Figure 31: Filter Specification corresponding to *Sensor*

$$
\begin{array}{l}
\textit{RobotArm\_}\textbf{filter} \\
\quad \textit{INIT} \\
\quad\quad status : ArmStatus \\
\quad\quad t : RealTime \\
\quad\quad \hline \\
\quad\quad mapping = \{status \mapsto t\} \\
\\
\quad ChangeStatus \\
\quad\quad status, status' : ArmStatus \\
\quad\quad t : RealTime \\
\quad\quad \delta t : Time \\
\quad\quad \hline \\
\quad\quad mapping = \{status \mapsto t, status' \mapsto t + \delta t\} \\
\\
\quad Pick \mathrel{\widehat{=}} [\, \delta t \geq t_{pi} \,] \wedge \\
\quad\quad [\, mapping = \{status \mapsto t, position \mapsto t, obj? \mapsto t + \epsilon_{1obj?}, newst? \mapsto t + \epsilon_{1newst?}\} \wedge \\
\quad\quad ChangeStatus[t / ChangeStatus.t] \wedge \\
\quad\quad obj?.ChangePosition[position/new?, t/obj?.ChangePosition.t] \wedge \\
\quad\quad obj?.ChangeStatus[t/obj?.ChangeStatus.t] \\
\quad Place \mathrel{\widehat{=}} [\, \delta t > t_{pl} \,] \wedge \\
\quad\quad [\, mapping = \{status \mapsto t, position \mapsto t, obj? \mapsto t + \epsilon_{1obj?}, newst? \mapsto t + \epsilon_{1newst?}\} \wedge \\
\quad\quad ChangeStatus[t / ChangeStatus.t] \wedge \\
\quad\quad obj?.ChangePosition[position/new?, t/obj?.ChangePosition.t] \wedge \\
\quad\quad obj?.ChangeStatus[t/obj?.ChangeStatus.t]
\end{array}
$$

Figure 32: Filter Specification corresponding to *RobotArm*

In the filter specification *RobotArm_***filter** shown in Figure 32, the filter *Pick* contains three sections: the first section asserts that the anticipated completion time of *Pick* operation must be at least equal to that of t_{pl} which is derived from the application domain. Based the specification of *Pick* operation, it also asserts that the completion time of *Pick* should be at least equal to the maximum of the completion times of the three component operations *ChangeStatus*, $obj?.ChangePosition$ and $obj?.ChangeStatus$. All the three component operations are invoked at the same time t at which the operation *Pick* is invoked since *Pick* is defined as a logical conjunction of the three operations.

The filter for *Place* can be interpreted in a similar way.

$$
\begin{array}{|l}
\hline
\textit{AsembledPart\_}\textbf{filter} \\
\quad \begin{array}{|l}
\hline
\textit{ChangePosition} \\
position, position' : Point \\
new? : Point \\
t : RealTime \\
\delta t, \epsilon_{1new?} : Time \\
\hline
mapping = \{position \mapsto t, position' \mapsto t + \delta t, \\
\qquad new? \mapsto t + \epsilon_{1new?}\} \\
\hline
\end{array} \\
\hline
\end{array}
$$

Figure 33: Filter Specification corresponding to *AssembledPart*

The description for the filter specification *AssembledPart* shown in Figure 33 is self-explanatory.

The filter *PerformAssembly* shown in Figure 34 asserts that the anticipated completion time of the operation *PerformAssembly* must be greater than or equal to the assembly time t_{as} and the output from this operation will appear only after t_{as} units of time.

$AssemblyPad$**_filter**

INIT

$status : Assembly\_Pad\_Status$
$t : RealTime$

$mapping = \{status \mapsto t\}$

$PerformAssembly$

$position : Point$
$status : Assembly\_Pad\_Status$
$cup?, dish? : Part$
$assembly! : AssembledPart$
$result! : Boolean$
$t : RealTime$
$\delta t, \epsilon_{1cup?}, \epsilon_{1dish?}, \epsilon_{2assembly!}, \epsilon_{2result!} : Time$

$$mapping = \{position \mapsto t, status \mapsto t,$$
$$cup? \mapsto t + \epsilon_{1cup?}, dish? \mapsto t + \epsilon_{1dish?},$$
$$assembly! \mapsto t + \epsilon_{2assembly!}, result! \mapsto t + epsilon_{2result!}\}$$
$$t_{as} \leq \epsilon_{2assembly!} \leq \delta t$$
$$t_{as} \leq \epsilon_{2result!} \leq \delta t$$

Figure 34: Filter Specification corresponding to *AssemblyPad*

$RotaryTable\_\mathbf{filter}$

INIT

$$
\begin{array}{l}
assembly\_end : Point \\
done : Boolean \\
t : RealTime \\
\hline
mapping = \{assembly\_end \mapsto t, done \mapsto t\}
\end{array}
$$

$Rotate$

$$
\begin{array}{l}
assembly\_end, assembly\_end' : Point \\
destination\_end, destination\_end' : Point \\
done, done' : Boolean \\
original? : AssembledPart \\
t : RealTime \\
\delta t, \epsilon_{1\,original?} : Time \\
\hline
mapping = \{assembly\_end \mapsto t, assembly\_end' \mapsto t + \delta t, \\
\quad destination\_end \mapsto t, destination\_end' \mapsto t + \delta t, \\
\quad done \mapsto t, done' \mapsto t + \delta t, \\
\quad original? \mapsto t + \epsilon_{1\,original?}\} \\
\delta t \geq t_{ao}
\end{array}
$$

Figure 35: Filter Specification corresponding to *RotaryTable*

The description of *RotaryTable*_**filter** given in Figure 35 is self-explanatory.

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

$PartsRecognitionSubsystem$ filter

INIT

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

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

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

PartArrived

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

Figure 36: Filter Specification corresponding to *PartsRecognitionSubsystem*

$PartsRecognitionSubsystem\_\mathbf{filter}(Contd...)$

$$
\begin{array}{l}
DivertPart\_RBC \mathrel{\widehat=} PartArrived[t/PartArrived.t] \wedge \\
\quad \left[mapping = \{parttype \mapsto t, s2 \mapsto t, dir? \mapsto t + \epsilon_{1dir?}\} \right] \wedge \\
\quad rbc.RotateNinety[t/rbc.RotateNinety.t] \wedge \\
\quad rbc.Start\_Belts\_RBC[t + rbc.RotateNinety.\delta t/rbc.Start\_Belts\_RBC.t] \wedge \\
\quad rbc.Move\_Belts\_RBC[t + rbc.RotateNinety.\delta t + rbc.Start\_Belts\_RBC.\delta t/ \\
\qquad\qquad rbc.Move\_Belts\_RBC.t] \wedge \\
\quad rbc.Stop\_Belts\_RBC[t + rbc.RotateNinety.\delta t + rbc.Start\_Belts\_RBC.\delta t+ \\
\qquad\qquad rbc.Move\_Belts\_RBC.\delta t/rbc.Stop\_Belts\_RBC.t] \wedge \\
\quad s2.ChangeStatus[t + rbc.RotateNinety.\delta t + rbc.Start\_Belts\_RBC.\delta t+ \\
\qquad\qquad rbc.Move\_Belts\_RBC.\delta t/s2.ChangeStatus.t] \\
\\
DivertPart\_RBD \mathrel{\widehat=} PartArrived[t/PartArrived.t] \wedge \\
\quad \left[mapping = \{parttype \mapsto t, s3 \mapsto t\} \right] \wedge \\
\quad rbc.Start\_Belts\_RBD[t/rbc.Start\_Belts\_RBD.t] \wedge \\
\quad rbc.Move\_Belts\_RBD[t + rbc.Start\_Belts\_RBD.\delta t/rbc.Move\_Belts\_RBD.t] \wedge \\
\quad rbc.Stop\_Belts\_RBD[t + rbc.Start\_Belts\_RBD.\delta t + rbc.Move\_Belts\_RBD.\delta t/ \\
\qquad\qquad rbc.Stop\_Belts\_RBD.t] \wedge \\
\quad s3.ChangeStatus[t + rbc.Start\_Belts\_RBD.\delta t + rbc.Move\_Belts\_RBD.\delta t/ \\
\qquad\qquad s3.ChangeStatus.t]
\end{array}
$$

Figure 37: Filter Specification corresponding to *PartsRecognitionSubsystem* (Contd...)

$$
\begin{array}{l}
\textit{PartsRecognitionSubsystem\_}\textbf{filter}(\textit{Contd...}) \\
\hline
DivertPart\_RBR \mathrel{\widehat=} PartArrived[t/PartArrived.t] \wedge \\
\quad \big[\, mapping = \{parttype \mapsto t, s2 \mapsto t, s3 \mapsto t, dir? \mapsto t + \epsilon_{1dir?}\} \,\big] \wedge \\
\quad rbc.RotateNinety[t + \epsilon_1 / rbc.RotateNinety.t] \wedge \\
\quad rbc.Start\_Belts\_RBR[t + rbc.RotateNinety.\delta t / rbc.Start\_Belts\_RBR.t] \wedge \\
\quad rbc.Move\_Belts\_RBR[t + rbc.RotateNinety.\delta t + rbc.Start\_Belts\_RBR.\delta t / \\
\qquad\quad rbc.Move\_Belts\_RBR.t] \wedge \\
\quad rbc.Stop\_Belts\_RBR[t + rbc.RotateNinety.\delta t + rbc.Start\_Belts\_RBR.\delta t + \\
\qquad\quad rbc.Move\_Belts\_RBR.\delta t / rbc.Stop\_Belts\_RBR.t] \wedge \\
\quad s4.ChangeStatus[t + rbc.RotateNinety.\delta t + rbc.Start\_Belts\_RBR.\delta t + \\
\qquad\quad rbc.Move\_Belts\_RBR.\delta t / s4.ChangeStatus.t] \\
\\
DivertPart \mathrel{\widehat=} DivertPart\_RBC[t/DivertPart\_RBC.t] \wedge \\
\quad DivertPart\_RBD[t/DivertPart\_RBD.t] \wedge \\
\quad DivertPart\_RBR[t/DivertPart\_RBR.t] \\
ResetBelts \mathrel{\widehat=} s5.ChangeStatus[t/s5.ChangeStatus.t] \wedge \\
\quad (\big[\, mapping = \{rbc \mapsto t, dir? \mapsto t + \epsilon_{1dir?}\} \,\big] \wedge \\
\qquad rbc.RotateNinety[t/rbc.RotateNinety.t]) \wedge \\
\quad (\big[\, mapping = \{rbc \mapsto t, dir? \mapsto t + \epsilon_{1dir?}\} \,\big] \wedge \\
\qquad rbc.RotateNinety[t/rbc.RotateNinety.t]) \\
\hline
\end{array}
$$

Figure 38: Filter Specification corresponding to *PartsRecognitionSubsystem* (Contd...)

The filter specification corresponding to *PartsRecognitionSubsystem* is described in Figures 36–38. 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.

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

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

Figure 39: Filter Specification corresponding to *AssemblySubsystem*(Contd...)

The filter specification corresponding to the class *AssemblySubsystem* is given in Figures 40. 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 *AssemblySubsystem_***filter** can be understood in a similar way.

4.3 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 logic of the filter specifications given in section 2.5. We show the application of the semantics for *AddPart* operation in Figure 40.

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. The resulting class is syntactically and semantically correct in Object-Z. Each functional variable will be augmented with the timing variable to which it is mapped in the filter specification. According to the logic foe Real-Time Object-Z, 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 40 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*?.

PartsRecognitionSubsystem

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

Figure 40: Timed Structure of *AddPart* in *Parts Recognition Subsystem*

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.

5 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.

5.1 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 *PartsRecognitionSubsystem*, 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 *PartsRecognitionSubsystem*, 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 *AssemblySubsystem*.

5.2 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 establish 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 41 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:

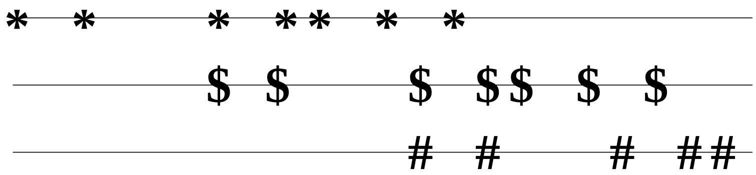
t_r	-	to recognize the dish
t_d	-	to travel from the camera to the rotary belt
t_{ro}	-	to rotate the rotary belt
$2 * t_d$	-	to travel from the start of the rotary belt to end of *ChanD*
t_{pi}	-	to be picked up by the right arm
t_{pl}	-	to place the dish on the assembly pad
t_{pl}	-	to place the cup 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



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 41: Events Chart for the Assembly Process

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

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.

5.2.1 An Abstract Assembly Program and its Verification

An algorithm to perform the assembly using the objects in the specification is given in Figure 42. 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 *PartsRecognitionSubsystem* and *AssemblySubsystem* 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. 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 $Op \mathrel{\widehat=} Op_1 \fatsemi Op_2$ is that sum of anticipated completion times of Op_1 and Op_2. We assume that the completion times for tasks such as assignment and logical checking are to be zero. For example,

$s2.ChangeStatus.\delta t = 0$

since it only involes one assignment. By this assumption, we derive the expressions based on only those timing parameters derived from the problem domain.

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

$$
\begin{array}{l}
PRS.AddPart[Cup/newpart?]\,{}_{9}^{\circ} \\
PRS.feeder.Start\,{}_{9}^{\circ} \\
PRS.feeder.Move[constant\_distance/dist?]\,{}_{9}^{\circ} \\
(PRS.RecognizePart \wedge PRS.AddPart[Dish/newpart?])\,{}_{9}^{\circ} \\
PRS.feeder.Start\,{}_{9}^{\circ} \\
PRS.feeder.Move[constant\_distance/dist?]\,{}_{9}^{\circ} \\
(PRS.DivertPart\_RBC\,{}_{9}^{\circ} \\
\quad (PRS.Resetbelts \wedge ASS.PickupCup) \\
) \wedge \\
(PRS.RecognizePart\,{}_{9}^{\circ} \\
\quad PRS.feeder.Start\,{}_{9}^{\circ} \\
\quad PRS.feeder.Move[constant\_distance/dist?]\,{}_{9}^{\circ} \\
\quad PRS.DivertPart\_RBD\,{}_{9}^{\circ} \\
\quad (PRS.ResetBelts \wedge ASS.PickupDish) \\
)\,{}_{9}^{\circ} \\
ASS.PlaceDish\,{}_{9}^{\circ} \\
ASS.PlaceCup\,{}_{9}^{\circ} \\
ASS.Assemble\,{}_{9}^{\circ} \\
ASS.RemoveAssembly
\end{array}
$$

Figure 42: An Assembly Algorithm

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

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

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

$$
\begin{aligned}
PRS.RecognizePart.\delta t &= max(PRS.s1.ChangeStatus.\delta t, PRS.feeder.Stop.\delta t, \\
&\qquad PRS.camera.Recognize.\delta t) \\
&= PRS.camera.Recognize.\delta t \\
&= t_r
\end{aligned}
$$

$$
\begin{aligned}
PRS.DivertPart\_RBC.\delta t = \;& max(PRS.PartArrived.\delta t,\\
& \quad PRS.rbc.RotateNinety.\delta t +\\
& \quad PRS.rbc.Start\_Belts.\_RBC.\delta t +\\
& \quad PRS.rbc.Move\_Belts\_RBC.\delta t,\\
& \quad PRS.s2.ChangeStatus.\delta t)\\
= \;& PRS.rbc.RotateNinety.\delta t +\\
& PRS.rbc.Start\_Belts.\_RBC.\delta t +\\
& PRS.rbc.Move\_Belts\_RBC.\delta t\\
\simeq \;& t_{ro} + 0 + 2 * t_d\\
\simeq \;& t_{ro} + 2 * t_d
\end{aligned}
$$

$$
\begin{aligned}
PRS.ResetBelts.\delta t = \;& max(PRS.s5.ChangeStatus.\delta t,\\
& \quad PRS.rbc.RotateNinety.\delta t)\\
= \;& PRS.rbc.RotateNinety.\delta t\\
= \;& t_{ro}
\end{aligned}
$$

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

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

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,

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

In the above expression, we have used the assumption that $t_{ro} < t_{pi}$ which is practically reasonable.

6 Conclusion

We have introduced a new notation called Real-Time Object-Z (RTOZ) for the specification and design of real-time reactive systems. RTOZ is a real-time extension of the specification language Object-Z. The novelty of our approach lies in decoupling the functional desciption from the timing constraints. The logic for RTOZ given in this report can be used to derive the timed system structure from these specifications. Due to the following reasons, we claim that our methodology is well suited for the development and maintenance of large and complex reactive systems:

- The Object Z notation achieves modularity in design. The functionality is easy to understand.
- Decoupling timing constraints from functional specification creates expressiveness of state abstractions, and modular timing specifications.
- Design is reusable in contexts where timing properties are not insisted.
- Inheritance hierachy can be defined independently in function description modules and time constraint modules to support multiple applications demanding different timing restrictions (functionalities) on the same functionality (timing constraint).

We have illustrated the application of our methodology by specifying a robot-based assembly cell using RTOZ. We have also proved safety and liveness properties of the assembly cell using the specifications. In [8], several specification notations have been employed to specify a similar problem. None of these methods seem to tackle both design complexity and formal verification. An exact analysis and formal verification of control systems based on the logic for RTOZ given in this report 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 *AssemblySubsystem* and *PartsRecognitionSubsystem*, this can be stated as:
The task

$$PRS.DivertPart\_RBC \mathbin{⨟} (PRS.ResetBelts \land ASS.PickupCup)$$

must be completed before the task

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

Let $S1 = PRS.DivertPart\_RBC \fatsemi (PRS.ResetBelts \wedge ASS.PickupCup)$ 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., $\quad t_{ro} + t_d + max(t_{ro}, t_{pi}) \leq t_r$

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

$$t_{ro} + t_d + t_{pi} \leq t_r$$

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

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.

References

[1] P.J. Antsaklis and K.M. Passino (Eds.), *An Introduction to Intelligent and Autonomous Control*, Kluwer Academic Publishers, 1993.

[2] 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.

[3] G. Booch, *Object-Oriented Analysis and Design with Applications*, (Second Edition), Benjamin Cummings Publications, 1994.

[4] R. Duke, G. Rose and G. Smith, "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.

[5] 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.

[6] 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.

[7] *IEEE Software*, Jan 1994.

[8] C. Lewerentz and T. Lindner (Eds.), *Formal Development of Reactive Systems*, Lecture Notes in Computer Science - 891, Springer-Verlag, 1995.

[9] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems - Specification*, Springer-Verlag, 1992.

[10] B.. Meyer, *Object-Oriented Software Construction*, Prentice Hall International Series in Computer Science, 1988.

[11] 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.

[12] J. Rumbaugh *et al.*, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.

[13] G. Smith, "A Logic for Object-Z", Technical Report 94-48, Department of Computer Science, University of Queensland, Australia, Dec 1994.

[14] G. Smith, "A Logic for Object-Z (Additional Rules)", Technical Report 95-24, Department of Computer Science, University of Queensland, Australia, Dec 1994.

[15] J.M. Spivey, *The Z Notation: A Reference Manual* (Second Edition), Prentice Hall International Series for Computer Science, 1992.

[16] S.G. Tzafestas (Ed.),*Intelligent Robot Systems*, Marcel Dekker Inc., NY, 1991.

[17] *The Z Notation*, ISO/IEC JTC 1/SC22 CD 13568, Sep 1995.

A Formal Verification

In this section, we show how the logic for RTOZ can be used to derive formal proofs. For simplicity, we take another example for this purpose. We give a real-time specification for a generalized version of the Ping Pong program taken from [9] and formally verify one of its properties.

A.1 Problem Statement

There are two modules M_1 and M_2 which are executed in parallel. Notice that in Object-Z (and hence in RTOZ), parallel operations indicate hand-shaking communication. Module M_1 checks for a global variable p which can assume only binary values. M_1 outputs $x = \overline{p}$[§]. Module M_2 receives x from M_1 and sets the value $y = x$ and outputs y. Module M_1 then receives y and sets $p = \overline{y}$. There are timing constraints on the variables p, x and y. The parallel execution is safe if for all times of execution τ, the value of p at time τ is equal to the value of p at time $\tau + \delta\tau$, $\delta\tau$ (> 0) being the anticipated completion time of the parallel execution.

[§] $\overline{p} = 0$ if $p = 1$; otherwise, it is 0.

A.2 Specification for the classes

$Binary == \{0, 1\}$

$$\begin{array}{|l}
\hline M_1 \\
\ldots \\
\quad \begin{array}{|l}
\hline Op_1 \\
p?, y?, x!, q! : Binary \\
\hline
x! = \textbf{ if } p? = 0 \textbf{ then } 1 \textbf{ else } 0 \\
q! = \textbf{ if } y? = 0 \textbf{ then } 1 \textbf{ else } 0 \\
\hline
\end{array} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline M_2 \\
\ldots \\
\quad \begin{array}{|l}
\hline Op_2 \\
x?, y! : Binary \\
\hline
y! = x? \\
\hline
\end{array} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline System \\
\quad \begin{array}{|l}
\hline
p : Binary \\
M_1 : Module_1 \\
M_2 : Module_2 \\
\ldots \\
\hline
\end{array} \\
Op \mathrel{\widehat=} M_1.Op_1[p/p?, p'/q!] \overset{\parallel}{\rightleftarrows} M_2.Op_2 \\
\ldots \\
\hline
\end{array}$$

A.3 Filter Specifications

$$\begin{array}{|l}
\hline M_1_\textbf{filter} \\
\quad \begin{array}{|l}
\hline Op_1 \\
p?, y?, x!, q! : Binary \\
t : RealTime \\
\delta t, \epsilon_{1p}, \epsilon_{1y}, \epsilon_{2x}, \epsilon_{2q} : Time \\
\hline
mapping = \{p? \mapsto t + \epsilon_{1p}, y? \mapsto t + \epsilon_{1y}, \\
\qquad x! \mapsto t + \epsilon_{2x}, q! \mapsto t + \epsilon_{2q}\} \\
\epsilon_{1p} = 0 \wedge \epsilon_{1y} > 0 \\
\hline
\end{array} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline M_2_\textbf{filter} \\
\quad \begin{array}{|l}
\hline Op_2 \\
x?, y! : Binary \\
t : RealTime \\
\delta t, \epsilon_{1x}, \epsilon_{2y} : Time \\
\hline
mapping = \{x? \mapsto t + \epsilon_{1x}, y! \mapsto t + \epsilon_{2y}\} \\
\epsilon_{2y} \geq \epsilon_{1x} \\
\hline
\end{array} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline System_\textbf{filter} \\
Op \mathrel{\widehat=} M_1.Op_1[p_t/p?, p'_{t+\delta t}/q!, t/M_1.Op_1.t] \wedge M_2.Op_2[t + M_1.Op_1.\epsilon_{2x}/M_2.Op_2.t] \\
\hline
\end{array}$$

A.4 Formal Verification

We now formally prove that the specifications satisfy the stated safety property.

Theorem 1 (Safety) *In System,* $Op.p'_{t+Op.\delta t} = Op.p_t$

<u>Proof:</u>
<u>Step 0:</u>

$$
\begin{aligned}
Op \quad = \quad & M_1.Op_1[p_t/p?, p'_{t+\delta t}/q!, t/M_1.Op_1.t] \wedge M_2.Op_2[t + M_1.Op_1.\epsilon_{2x}/M_2.Op_2.t] \ \rightrightarrows \\
& M_1.Op_1[p/p?, p'/q!] \ \overset{\parallel}{\rightleftarrows} \ M_2.Op_2
\end{aligned}
$$

...System and System_**filter**

$$
\begin{aligned}
= \quad & \big[\, p, p', y?, x! : \mathit{Binary} \mid \\
& x!_{M_1.Op_1.t+M_1.Op_1.\epsilon_{2x}} = \ \mathbf{if} \ p_{Op.t} = 0 \ \mathbf{then} \ 1 \ \mathbf{else} \ 0 \wedge \\
& p'_{t+\delta t} = \ \mathbf{if} \ y?_{M_1.Op_1.t+M_1.Op_1.\epsilon_{1y}} = 0 \ \mathbf{then} \ 1 \ \mathbf{else} \ 0 \wedge \epsilon_{1p} = 0 \wedge \epsilon_{1y} > 0 \,\big] \ \overset{\parallel}{\rightleftarrows} \\
& \big[\, x?, y! : \mathit{Binary} \mid y!_{M_2.Op_2.t+M_2.Op_2.\epsilon_{2y}} = x?_{m_1.Op_2.t+M_2.Op_2.\epsilon_{1x}} \wedge \epsilon_{2y} \geq \epsilon_{1x} \,\big]
\end{aligned}
$$

...ClassFilterBinding and ApplyFilter

We now prove the timing sequences individually and apply them to the operations. This will simplify the proof steps:

From Step 0, infer the following:

Step 1	$Op.t = M_1.Op_1.t$	Parallel RightLeft
	$Op.t < M_1.Op_1.t + M_1.Op_1.\epsilon_{2x}$	addition
Step 2	$M_2.Op_2.t + M_2.Op_2.\epsilon_{1x} = M_1.Op_1.t + M_1.Op_1.\epsilon_{2x} + M_2.Op_2.\epsilon_{1x}$	System_**filter**
Step 2.1	$M_1.Op_1.t + M_1.Op_1.\epsilon_{2x} \leq M_2.Op_2.t + M_2.Op_2.\epsilon_{1x}$	Step 2, Timing Parameters
Step 3	$M_2.Op_2.t + M_2.Op_2.\epsilon_{1x} < M_2.Op_2.\epsilon_{2y}$	M_2_**filter**
Step 4	$M_2.Op_2.t + M_2.Op_2.\epsilon_{2y} = M_1.Op_1.t + M_1.Op_1.\epsilon_{1y}$	Parallel RightLeft
Step 5	$t + Op.\delta t = t + max(M_1.Op_1.\delta t, M_2.Op_2.\delta t + M_1.Op_1.\epsilon_{2q})$	Parallel RightLeft
Step 5.1	<u>Case 1:</u> $t + Op.\delta t = t + M_1.Op_1.\delta t$	
Step 5.1.1	$t + Op.\delta t \geq t + M_2.Op_2.\delta t + M_1.Op_1.\epsilon_{2q}$	Step 5
Step 5.1.2	$t + Op.\delta t \geq t + M_2.Op_2.\epsilon_{2y} + M_1.Op_1.\epsilon_{2q}$	Timing Parameters
Step 5.1.3	$t + Op.\delta t \geq t + M_1.Op_1.\epsilon_{1y}$	Parallel RightLeft
Step 5.2	<u>Case 2:</u> $t + Op.\delta t = t + M_2.Op_2.\delta t + M_1.Op_1.\epsilon_{2q}$	
Step 5.2.1	$t + Op.\delta t \geq t + M_1.Op_1.\delta t$	Step 5
Step 5.2.2	$t + Op.\delta t \geq t + M_1.Op_1.\epsilon_{1y}$	Timing Parameters

We now compute the values for the variables in *Op* according to their histories:

Step 6 **from** Steps 0, 1 and $History(p)$
infer $M_1.Op_1.x! = M_1.Op_1.\overline{p}$
Step 7 **from** Steps 0, 2, 6 and $History(x)$
infer $M_2.Op_2.x? = M_1.Op_1.x! = M_1.Op_1.\overline{p}$
Step 8 **from** Steps 0, 3 and 7
infer $M_2.Op_2.y! = M_2.Op_2.x? = M_1.Op_1.\overline{p}$
Step 9 **from** Steps 0, 4, 8 and $History(y)$
infer $M_1.Op_1.y? = M_2.Op_2.y! = M_1.Op_1.\overline{p}$
Step 10 **from** Steps 0, 5 and 9
infer $M_1.Op_1.q! = M_1.Op_1.\overline{y!} = M_1.Op_1.p$
Step 11 **from** Steps 0 and 10
infer $Op.p'_{t+\delta t} = M_1.Op_1.q! \wedge Op.p_t = M_1.Op_1.p$
infer $Op.p'_{t+\delta t} = Op.p_t$