The Formal Semantics of Filters in the Specification of Real-Time Systems

K. Periyasamy and V.S. Alagar[†]

Technical Report TR-02-01

Department of Computer Science
University of Wisconsin-La Crosse
La Crosse, WI 54601
U.S.A.

[†]Department of Computer Science, Concordia University, Montreal, Canada.

Abstract

This paper presents the specification language *Real-Time Object-Z* (RTOZ), a conservative extension of the Object-Z specification language, for developing specification of timed systems. RTOZ has two tiers, the first tier for specifying the functional behavior of an untimed system, and the second tier for specifying timed properties of the system that is specified in the first tier. Both the tiers use the syntax of the Object-Z language. The logic of RTOZ, obtained as an extension of the logic of Object-Z, provides the basis for constructing timed system specification from the specifications in the two tiers of RTOZ. The logic formally specifies the intended meaning of timed objects and RTOZ operations, thus providing a formal semantics of RTOZ. The expressive power of the language is illustrated with the specication of a network mailer example.

1 Introduction

Real-time concurrent systems and reactive systems are in general large complex systems to design and to analyze. For instance, telecommunication networks, air traffic control systems and automated parts recognition systems are some example of real-time systems that have complex functional and timing requirements. Some of the real-time systems are safety-critical and hence rigorous analysis of their requirements is mandatory in order to ensure safety and liveness properties. The requirements of a system can be formally specified using a formal specification language to conduct such a rigorous analysis. Recently, formal specification languages found wide-spread applications in software development, particularly in requirements engineering.

A formal analysis of requirements can identify ambiguities and inconsistencies in the requirements. Software developers are also interested in deriving a design from the requirements that is easily manageable and implementable. The object-oriented approach is widely used for software design because of its inherent advantages such as easy maintenance and support for reuse. Combining with a formal approach, an object-oriented formal specification language provides a number of advantages: (i) Being formal, a requirements specification written in such a language will identify the ambiguities and inconsistencies in the specification. (ii) Using the object-oriented principles, a modular specification can be developed and hence the developer can focus on individual modules of the specification. (iii) The object-oriented approach provides a smooth transition from requirements to design and then to implementation. This facility can be exploited to refine the specification towards a correct design and a verifiable implementation.

This paper presents a real-time specification language called *Real-Time Object-Z* (RTOZ), that is a conservative extension of Object-Z specification language. RTOZ has two tiers, the first tier for specifying the functional behavior of an untimed system, and the second tier for specifying the timed properties of the system that is specified in the first tier. The second tier consists of a set of *filters* that are mapped to the operations in the first tier. These filters carry the timed variables and their mapping to the functional variables in the operations that are specified in the first tier. RTOZ uses the syntax of the Object-Z language in both the tiers. The timed specification of a system is obtained by *applying* filters in the second tier to their corresponding operations in the first tier. The semantics of RTOZ also includes the composition of filters that correspond to the composite operations in the first tier. The rules for composing filters are given in axiomatic style. The logic used in the axiomatic descriptions of RTOZ is an extension of the logic used by Smith [18] to provide the semantics of Object-Z. Thus, the rules formalize the meaning of intended operations in RTOZ.

1.1 Related Work and Significance of RTOZ Design

Wirth [22] argued that a real-time programming language should be a simple extension of a non-real-time programming language with additional syntax for stating time constraints. When this approach was applied in a naive manner to object-oriented programming languages, anomalies due to inheritance and polymorphism arose in the language. To correct these flaws, real-time filters [1] were introduced which are real-time extensions of object-oriented programming constructs. RT-Synchoronizers [15] is another high-level programming construct for specifying time constraints in a distributed concurrent system. Both real-time filters and RTSynchronizers are programming structures and are at a lower level of abstraction compared to the abstraction level in a specification language. At the specification layer it is much easier to reason about timed properties such as consistency, safety and liveness, and hence is easier to correct design flaws before deploying the system.

Wirth's observation for the design of real-time languages has been adopted and applied to real-time specification languages as well. For instance, timed automata [4] are automata augmented with continuous variables, called *clocks*, that can be tested and updated during transitions. The parallel composition of timed automata is a timed automaton such that the invariants associated with a state in the product automaton is the conjunction of the invariants associated with the component states. Timed CSP [16] and Synchronous Calculus of Communicating Systems (SCCS) [9] are the conservative extensions of untimed CSP [10] and CCS [13] respectively. Timed CSP introduces the timing primitives *delay* and *timeout* to capture when and how long a process will wait for a certain event to occur and the maximum time limit for a process execution. SCCS models time using integer domain and uses a global clock for synchronization of processes. Each action in SCCS is augmented with the time at which the action occurs. The primitive modeling unit in Timed CSP and SCCS is a process which is an invidivisible atomic action. In Timed CSP, a process can be augmented with data variables that represent both communicating variables (input and output variables of the process) as well as internal variables of the process. However, a process in SCCS can only include communicating variables and not internal variables. Mahony and Hayes [12] extended the Z formal notation [20] with additional syntax to capture discrete and continuous real-time information. The Quartz method described in [7] is another attempt to extend the Z notation with real-time primitives. The design of RTOZ differs from these extensions of Z in two important ways: object-orientation, and compositionality.

Timed action systems are extensions of untimed action systems [6], a formalism for representing concurrent behaviors based on interleaving semantics. To give a timed interpretation of an untimed action system, the syntax of the untimed system is extended to include timed predicates for guards and actions. The timed system can model time-consuming pre-emptive actions. Guards and actions involve both time and functional variables and Z language is used to specify the predicates. The problem with timed action systems is that the operational behavior of Z operations have to be interpreted carefully to reflect concurrency, and refinements are hard to perform. Mahony and Dong [11] have described Timed Communicating Object-Z (TCOZ) obtained by combining Object-Z and Timed CSP. The motivation behind TCOZ, according to its authors, is that Timed CSP does not capture state-based information of a real-time system and hence Object-Z is blended with Timed CSP to capture both state-based and time-based information. In contrast, RTOZ has a uniform syntax, the syntax of Object-Z, separates functionality from time domain, and provides structuring in specification description. Refinements can be done independently on either tier of RTOZ.

TRIO+ [14] is an object-oriented real-time specification language and is a conservative extension of TRIO. Both TRIO and TRIO+ are based on temporal logic and use quantitative notions of time by providing a metric to timing primitives. The formal semantics of TRIO is claimed to accommodate a variety of time structures including dense, discrete and finite structures [8]. RTOZ differs from TRIO+ and other object-oriented specification languages in the following ways:

- RTOZ uses a discrete notion of time. Time is observed from a global clock. The passage of time is not explicitly modeled, but its effect is observable in the *History* associated with data variables. The instant at which a variable changes its value is implementation dependent and not observable, but the expected time at which it's effect is to be come available can be specified. This is the rationale behind the decision not to specify changes to auxiliary time variables.
- RTOZ language has two tiers, one for the functional specification of a system, and the other for specifying filters involving time constraints. Object-Z notation is used to specify both tiers. Rules are provided for composing filters and applying filters to their functional components. Hence, learning and using RTOZ is as easy as using Object-Z.
- The separation of functional and timing behaviors and the overlaying semantics of RTOZ filters suggest further extensions of the language towards multi-dimensional specifications. For example, to specify time-dependent business transactions, such as E-commerce, one could add a third tier to RTOZ in which business rules are specified. The existing semantics can be extended by introducing rules for composing business filters in the third tier, and for applying business filters to RTOZ specifications.

2 The Notation for RTOZ

A specification in RTOZ has two tiers: one tier for functionalities and the other for timing constraints. Each tier uses the syntax and semantics of Object-Z. The tier where timing constraints are stated introduces *filter specifications*. The notation for Object-Z, as given in [5, 19], and the schema calculus for Z as given in [20] are followed. Some familiarity with Object-Z language is assumed of the reader. Only a brief description of the language is given below.

2.1 Object-Z Notation

A specification in Object-Z consists of global definitions and class definitions. The global definitions are shared by all classes in the specification. Global definitions and the paragraphs within each class definition are given using the Z notation. The structure of a class definition follows:

$$
\begin{array}{|l}
\hline
\textit{ClassName} \\
\upharpoonright(\textit{visibility list}) \\
\textit{list of inherited class names} \\
\textit{type definitions} \\
\textit{constant definitions} \\
\textit{state space schema} \\
\textit{initial state space schema} \\
\textit{operation schemas} \\
\hline
\end{array}
$$

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 exported. The *list of inherited class names* of C includes 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. *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* introduces the attributes of the class. The attributes are divided into primary and secondary variables. The primary variables can be changed by an operation only if the operation declares them in its Δ-list (definition of Δ-list is given below). The secondary variables are generally computed from primary vairables. The intention to keep a set of secondary variables is to maintain some information that can be derived from the primary variables and made readibly available for further access. The *initial state space schema* describes the set of all initial states of objects that are instantiated from the class.

Operation schemas are defined in the same way as in Z, with one exception. While an operation in Z includes the state spaces that are affected by the operation using the Δ or Ξ notation, the state space is implicit for an operation in Object-Z, namely the state space of the class. The notation $\Delta(x, y)$ in an operation Op in Object-Z (called the Δ-list of the operation) asserts that the state variables x and y are modified by Op; other state variables are left unmodified 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 five composition operators in Object-Z. These are: *conjunction* ($\wedge$), *parallel composition* ($\|$), *sequential composition* ($\fatsemi$), *choice operator* ($[]$) and *environment enrichment operator* ($\bullet$). 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. The 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; 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 pre-conditions are satisfied, the choice is non-deterministic. The environment enrichment operator is used to extend the signature of one operation to another operation. Accordingly, in the expression $Op_1 \bullet Op_2$, the operation Op_2 can access the variables in the signature of the operation Op_1.

2.2 Filter Notation

RTOZ provides two types with regard to time: *RealTime* and *Time*. The carrier set of *Real-Time* is the set of all instances of a global clock; *Time* represents the metric notion of time. Filter specifications in the second tier of RTOZ may use only these two types of time.

A *filter specification* consists of a collection of filter classes. Each filter class corresponds to exactly one class in the first tier of RTOZ. A filter class consists of a collection of filters, each filter corresponding to exactly one operation in the corresponding class in the first tier. Figure 1 shows a schematic view of classes, operations and filers in an RTOZ specification. The state schema of a class is copied as the state schema of the corresponding filter class. The signature of the operation schema is copied into the signature of the filter corresponding to the operation. The following timing variables are introduced in the signature of each filter:

- t (of type $RealTime$) denotes the invocation time of the operation;
- δt (of type $Time$) denotes the anticipated completion time of the operation;
- for each input variable $x?$ in the operation , $\epsilon_{1x?}$ (of type $Time$) denotes the offset from the invocation of the operation at which the variable $x?$ receives an input value; and
- for each output variable $y!$ in the operation, $\epsilon_{2y!}$ (of type $Time$) denotes the offset from the invocation of the operation at which the operation outputs the value $y!$.

A filter has the same name as that of the operation to which it corresponds. The following naming coventions are followed throughout the paper:

- The filter specification of the class C is named C**_filter**. For example, the filter specification of the class *Mail* will be named *Mail*_**filter**.
- The filter corresponding to the operation Op is denoted as $\overline{Op}$.

The predicate part of a filter schema includes the predicate

timemap = $\{ \ldots \}$

This predicate describes the mapping of untimed variables to the timing variables that are declared in the signature of the filter schema. Additional timing constraints involving global variables may also be introduced in the predicate part of the filter schema.

The following example shows an operation in Object-Z, its filter in Object-Z syntax, and the *application* of the filter to the operation:

<u>Global declaration:</u>

Let t_g denote a global timing variable in an application.

$$t_g : Time$$

<u>Untimed Operation - First Tier of RTOZ:</u>

Op

$\Delta(y)$
$p? : \mathbb{N}$

$x < p? \Rightarrow y' = p?$
$x \geq p? \Rightarrow y' = y$

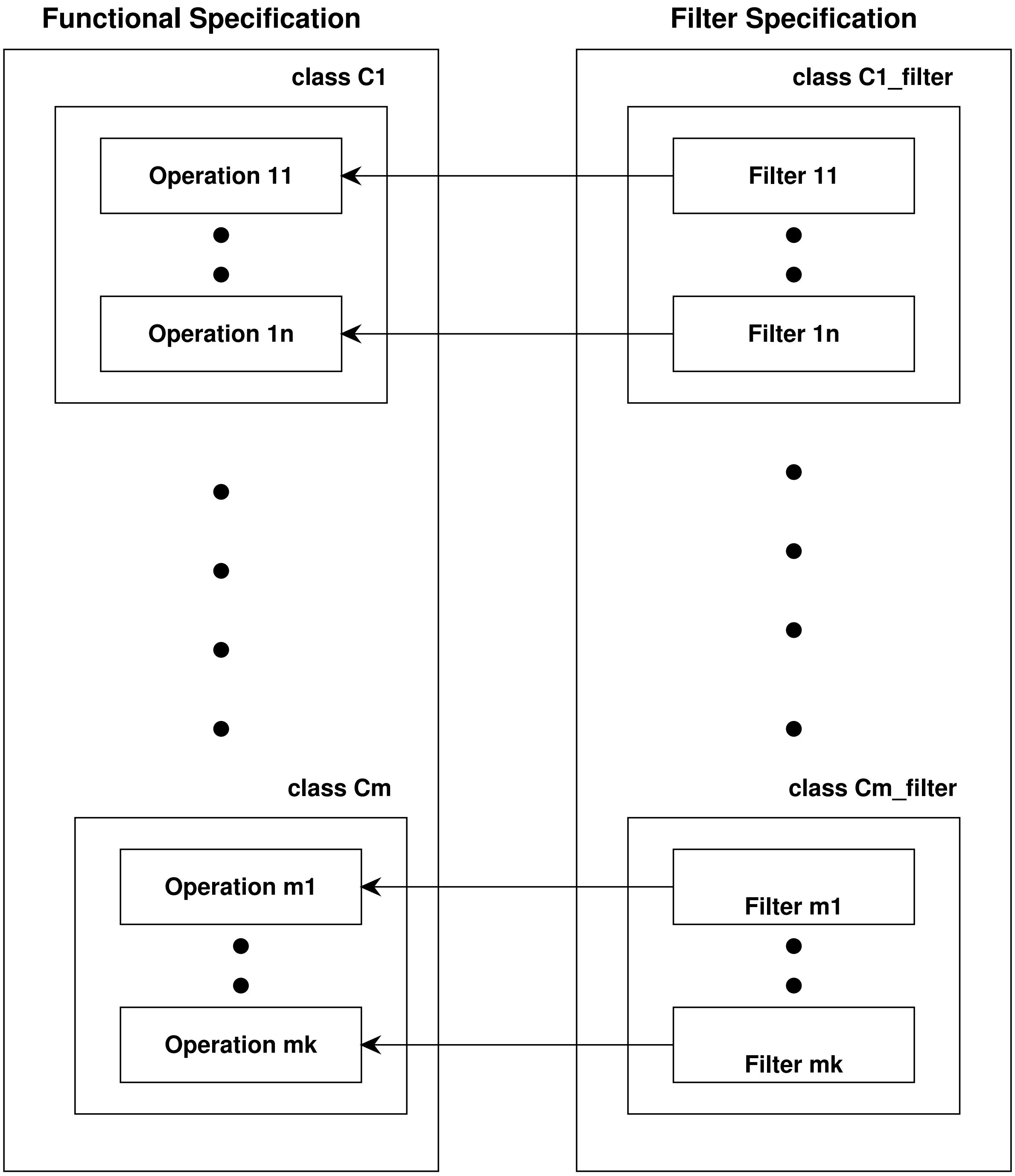

Figure 1: Classes, Operations and Filters in an RTOZ Specification

The variables x and y are assumed to be defined in the state space of the class in which Op is introduced as an operation.

Filter for Op - Second Tier of RTOZ:

$$
\begin{array}{|l}
\hline
\overline{Op} \\
p? : \mathbb{N}; \\
t : RealTime; \\
\delta t, \epsilon_{1p?} : Time \\
\hline
\textbf{timemap} = \{x \mapsto t, y \mapsto t, p? \mapsto t + \epsilon_{1p?}, y' \mapsto t + \delta t\} \\
\delta t \leq t_g \\
\hline
\end{array}
$$

In the filter $\overline{Op}$, the constraint $\delta t \leq t_g$ is an additional timing constraint.

The result of applying the filter $\overline{Op}$ to the operation Op is a timed specification of the operation, which is given next.

Result of Applying Filter $\overline{Op}$ to Operation Op:

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

Operation schema TOp gives the timed behavior of the untimed operation Op subject to the constraints stated in the filter $\overline{Op}$. The signature of TOp is obtained by conjoining the signatures of Op and its filter. The predicate part of TOp includes the predicate part of Op for which the mapping from the filter is applied, and the additional constraint given in the filter $\overline{Op}$. Variables such as x_t that are suffixed by t denote the respective values of the variables at time t.

3 Timed Objects and Filters

This section introduces the operators, meta functions and the style used in describing the semantics of RTOZ. It also includes a formal description of the model of time used by RTOZ and a formal semantics for the filters.

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.

$\leadsto$ is a binary operator and denotes the application of time maplets (functional variables mapped to time variables) to data entities (variables, expressions, objects etc.).

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. When applied to a filter class, it returns the names of all filters in that class.

$\beta_?$ and $\beta_!$ respectively return the basenames of inputs and outputs of an operation. For example, $x?$ is an input variable and x is its basename.

ρ 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.

ω returns the name of the operation or the name of the filter that is passed as parameter.

Π_c returns the name of a class. The function Π_{str} returns the name of a filter class after stripping off the keyword **_filter** in the name of the filter class.

α_T returns the set of names of timing variables involved in a filter. It includes the timing variables declared in the filter and the global timing variables used in the predicate part of the filter.

Υ returns the mapping of functional variables to timing variables in a filter.

Every filter must contain a predicate of the format $\mathbf{timemap} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. The function Γ_t extracts this predicate from the predicate part of the filter schema. That is,

$\Gamma_t\ (\overline{Op})$ = "**timemap** = $\Upsilon\ (\overline{Op})$"

Γ_r returns the predicates excluding the **timemap** predicate in a filter.

$\mathcal{D}$ returns the declaration of an operation or a filter. It includes state variables, Δ 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: $\mathcal{P} = \Gamma_t \wedge \Gamma_r$.

$VALUE[x, \tau]$ returns the value of x at time τ, where x is any data object.

$last$ accepts a data object x and a time τ and returns 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.

Several parts of the semantics are given as inference rules; an inference rule takes the following format:

Rule: (*Rule name*)

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

The premises are given as a list of sequents:

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

and the conclusion is given as a single sequent:

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

The $Proviso$ is a predicate in conjunctive normal form. The $Conclusion$ can be derived for a given $Premises$ only when the $Proviso$ is true. In writing the $Proviso$, line breaks between two predicates is assumed to be 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). In the sequent

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

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

3.1 Model of Time

The type $RealTime$ and $Time$ are coercive with respect to addition and subtraction. The following rules describe the allowable operations on variables of type $RealTime$ and $Time$, and the type of the result in each case.

Rule: (Coersion)

$$\frac{r, s : RealTime \vdash \quad t_1, t_2 : Time \vdash}{\vdash (r \pm s) : RealTime \wedge (r \pm t_1) : RealTime \wedge (t_1 \pm t_2) : Time}$$

There exists a fixed time t_0 of type *RealTime* that denotes the initialization time of the system. Every variable t of type *RealTime* will be greater than or equal to t_0. All variables of the filter

$INIT$ in the filter class 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. However, the $INIT$ filters in individual classes may have their own initiation time. All global variables are augmented with t_0.

The formal model of timed objects 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,

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

The datatype seq_0 has the definition

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

The predicate part of the formal definition for *History* asserts that the history of a data object x is recorded at monotonically increasing time points and the value of the data object does not change between any two consecutive time points. 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 $VALUE[x, \tau]$. Given the history of the data object x, the following rule asserts that the last value in $History(x)$ denotes the most recent value of x.

$$\frac{History(x) \vdash}{\vdash x_\tau = VALUE[x, History(x)(\#History(x) - 1)]} \quad [\, p \,]$$

where $p \equiv \tau \geq History(x)(\#History(x) - 1)$

It is assumed that $History$ is recorded by the program and so at the specification level no modifier operation for $History$ is provided.

3.2 Semantics of Timed Objects

This section describes the meaning of augmenting a functional variable v with a timed variable t (i.e., v_t):

[‡]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.

- If v is a simple variable (of type $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$), then $v_t = VALUE[v, t]$.
- If v denotes an object of a class C, then unless otherwise specified, every attribute of C will be augmented with t and the semantics is separately applied to each attribute. Since each attribute of C can be individually identified, it is also possible to augment different variables with different times.
- If v denotes a set, then every element of the set is augmented with the same time t. The interpretation is that all elements of the set are observed at the same time t, no matter at what time the individual elements are created. Moreover, after the application of the filters, the sets created using timed objects preserve the behavioral properties of sets (namely, union, intersection and difference).
- If v denotes a tuple, 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 acceptable to assign different timing variables to the components of a tuple as in $v_t = (p_{t_1}, q_{t_2})$ where $t_1 \leq t$ and $t_2 \leq t$. The same semantics applies to a variable that denotes a sequence.
- If v denotes a function and is defined using the axiomatic declaration, then no timing variable is associated with v. The justification comes from the fact that Object-Z (and Z) treats axiomatic descriptions as constants. When invoking v, the parameters of v can be instantiated with timed variables.
- If v denotes a schema, then unless otherwise specified, the values of all the variables that are declared within v and are accessible within v, are augmented with t. As in the case of objects, it is permissible to augment the individual variables within v with different times.
- If v denotes a constant, then the value of the constant remains unaltered over time.

3.3 Applying Timemap

The application of **timemap** defined in a filter $\overline{Op}$ to its corresponding operation Op involves the time map application to variables, expressions, and predicates. The result of application is a new schema in which the signature part is the union of the signatures of the operation and the filter, and the timed predicate part is obtained according to the rules defined below.

3.3.1 Applying Timemap to Expressions

The notation $M \rightsquigarrow E$ denotes the substitution of the map M to the expression E. Table 1 gives a list of rules governing the application of a time map to an expression.

3.3.2 Applying Timemap to Predicates

The substitution of **timemap** into predicates is very similar to the substitution of **timemap** into expressions. The notation $M \rightsquigarrow P$ denotes the substitution of the map M into a predicate P. Rules in Table 2 describe the application of M to predicates.

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

Table 1: Substitution of time-variables mapping into expressions

3.4 Semantics of Filters

The rule "Filter" given below formally describes the relationship between the four timing parameters t, δ, ϵ_{1x} and ϵ_{2y} that were discussed in Section 2.2.

Rule: (Filter)

$$\frac{Op = \{d_1 \mid p_1\} \vdash \qquad \overline{Op} = \{d_2 \mid p_2\} \vdash}{\begin{array}{l} \vdash \delta t > 0 \wedge (0 \leq \epsilon_{1x} < \delta t) \wedge (0 < \epsilon_{2y} \leq \delta t) \\ \omega(Op) = \omega(\overline{Op}) \\ \mathcal{D}(\overline{Op}) \setminus \alpha_T(\overline{Op}) \subseteq \mathcal{D}(Op) \\ p_2 \Rightarrow \Gamma_t(\overline{Op}) \end{array}} \quad [\ p\ ]$$

where, $p \equiv x \in \beta_?(Op) \wedge y \in \beta_!(Op)$
$\qquad (t : RealTime) \in d_2 \wedge (\delta t, \epsilon_{1x}, \epsilon_{2y} : Time) \in d_2 \wedge$

4 Semantics of RTOZ

A formal description of RTOZ semantics includes the formal semantics for binding a filter specification with a class specification, the semantics for applying a filter to its operation, and the semantics for composing filters corresponding to composite Object-Z operations.

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.
$M \rightsquigarrow (P \wedge Q) = (M \rightsquigarrow P) \wedge (M \rightsquigarrow Q)$	Application to conjunction.
$M \rightsquigarrow (P \vee Q) = (M \rightsquigarrow P) \vee (M \rightsquigarrow Q)$	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 free variables in 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 free variables in 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 applied to a predicate.

Table 2: Substiution of time-variables mapping into predicates

4.1 Binding a Filter Specification with its Class Specification

The filter class $C\_\mathbf{filter}$ corresponding to a class C contains a collection of filters that correspond to the operations in C. The result of binding $C\_\mathbf{filter}$ with C creates a timed class definition C_r. The formal semantics given below describes this binding. The notation $\rightrightarrows$ denotes the binding of $C\_\mathbf{filter}$ to C and the notation $\hookrightarrow$ denotes the application of a filter $\overline{Op}$ to its operation Op.

<u>Rule:</u> (ClassFilterBinding $\rightrightarrows$)

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

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

The rule 'ClassFilterBinding' asserts that Op must be an operation from the class C, $\overline{Op}$ is its filter derived from the filter class $C\_\mathbf{filter}$ and OP is the timed operation derived by applying $\overline{Op}$ to Op. The formal semantics of filter application ($\hookrightarrow$) is given in the next section. The $INIT$ filter of $C\_\mathbf{filter}$ is also applied in a similar way to the $INIT$ schema of C.

4.2 Applying Filters

When a filter is applied to an operation, the result is a timed specification in which the functional variables of the operation are augmented with the timing variables in the filter. The following is a list of constraints on the application of filters:

- The timing variables derived from the application domain are declared globally outside the filter specifications and can be accessed by all the filter specifications.
- For a class C, there can be at most one filter class; this filter class is named as $C\_\mathbf{filter}$.
- For an operation Op in C, there can be at most one filter $\overline{Op}$ in $C\_\mathbf{filter}$; this filter has the same name as Op.
- The filter class $C\_\mathbf{filter}$ contains only the filters that correspond to the operations of C.
- A filter $\overline{Op}$ is defined as a schema. The signature of $\overline{Op}$ contains a set of timing variables and a set of functional variables; these functional variables must have been defined in the signature of the operation Op.
- A filter $\overline{Op}$ defines the mapping between the functional variables and timing variables that are available in the declaration of $\overline{Op}$. Besides the mapping, $\overline{Op}$ can introduce additional constraints between timing variables. However, it cannot introduce new functional variables.
- A filter $\overline{Op}$ cannot be invoked independently; it can only be applied to the operation Op. In a similar way, an operation Op must be invoked along with its filter $\overline{Op}$, if the latter exists.
- The $INIT$ schema is treated similar to operations and hence it has its own filter named $\overline{INIT}$. Other constructs within a class definition such as *visibility list, type definitions* and *state space schema* are not affected by the filter specification.

Filter application is denoted by $\hookrightarrow$. Formally,
<u>Rule:</u> (ApplyFilter $\hookrightarrow$)

$$\frac{Op = \{d_1 \mid p_1\} \vdash \qquad \overline{Op} = \{d_2 \mid p_2\} \vdash}{\overline{Op} \hookrightarrow Op \vdash OP = \{d_3 \mid p_3\}} \quad [\, p \,]$$

where $p \equiv \omega(OP) = \omega(Op) = \omega(\overline{Op})$
$\mathcal{D}(\overline{Op}) \setminus \alpha_T(\overline{Op}) \subseteq \mathcal{D}(Op)$
$d_3 = d_1 \cup d_2$
$p_3 = (\Upsilon(\overline{Op}) \rightsquigarrow p_1) \wedge \Gamma_r(\overline{Op})$

4.2.1 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. A client invoking an operation of a class C must provide values for the starting time t, and the offsets $\epsilon_{1x?}$ for every input variable in Op. The completion times and offsets for output variables will be determined by the implementation.

Rule: (OperationInvocation)

$$\frac{\vdash a \in C \quad C ::| \; Op \in \Omega(C) \vdash \quad C\_\mathbf{filter} ::| \; \overline{Op} \in \Omega(C\_\mathbf{filter}) \vdash}{\begin{array}{l}\vdash a.Op = \big[\, a \circledast \mathcal{D}(Op) \cup a \circledast \mathcal{D}(\overline{Op}) \mid \\ \quad a \circledast (\Upsilon(\overline{Op}) \rightsquigarrow \mathcal{P}(Op) \wedge \Gamma_r(\overline{Op}) \wedge x_1' = x_1 \wedge \ldots x_n' = x_n) \,\big] \\ \qquad \big\backslash (y_1, \ldots, y_m, y_1', \ldots, y_m')\end{array}} \; [\, p \,]$$

where $p \equiv \omega(Op) = \omega(\overline{Op})$
$\alpha_1(C) \cup \alpha_1(C\_\mathbf{filter}) \setminus \rho\,(C, Op_1) = \{x_1, \ldots, x_n\}$
$\alpha_1(C) \cup \alpha_1(C\_\mathbf{filter}) \cup \alpha_2(C) \cup \alpha_2(C\_\mathbf{filter}) = \{y_1, \ldots, y_m\}$
$\phi(Op) \subseteq \alpha(C)$
$\phi(\overline{Op}) \subseteq \alpha(C\_\mathbf{filter})$

4.3 Semantics for Composite Filters

For simplicity in exposition the names of Object-Z composite operators are used in both tiers of RTOZ. That is, if a composite operator $\theta \in \{\,{}_{9}^{\circ}\,, \bigwedge, \|, \; [] \;, \bullet\}$ is used to construct a composite operation then the same symbol is also used to denote the composite operator for constructing the filter of the composite operation from the filters of the component operations. Such an overloading of operator names should not cause any confusion because of the semantic difference. Thus, for example, the filter of the composite operator $Op = Op_1 \ {}_{9}^{\circ} \ Op2$ is $\overline{Op} = \overline{Op_1} \ {}_{9}^{\circ} \ \overline{Op_2}$. A filter application satisfies the equation

$$(\overline{Op_1}\theta\overline{Op_2}) \hookrightarrow (Op_1\theta Op_2) \equiv (\overline{Op_1} \hookrightarrow Op_1) \;\theta'\; (\overline{Op_2} \hookrightarrow Op_2)$$

that defines the semantics of the composite operation θ' for timed operations. The semantics describes the rules for constructing composite filters. These rules combined with the rules for filter binding and filter application provide a justification of the above equation. The semantics for composite filters describes (i) how the timing parameters of a composite filter are computed from the timing parameters of the component operations, and (ii) how the time map of the composite filter is constructed. A composite operation is invoked only after its filter is constructed.

Input and output variables in an operation are locally defined. If necessary, input and output variables in the operations participating in the composition may be renamed so that in the composite operation, all variable names are distinct.

Sequential Composition Let $Op = Op_1 \, {}_{9}^{\circ} \, Op_2$. 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 $t + \overline{Op_1}.\delta t$. The completion time of Op is the sum of the completion times of Op_1 and Op_2. The input and output variables of Op are the union of the input and output variables of Op_1 and those of Op_2. The timing parameters for the input and output variables in $\overline{Op}$ that are derived from $\overline{Op_1}$ will not change in $\overline{Op}$, while the timing parameters that are derived from $\overline{Op_2}$ will be modified. This is due to the shift in the invocation time of Op_2.

It is sufficient to consider two categories of variables in $\overline{Op}$ for the construction of its time map: variables that are not shared by Op_1 and Op_2, and variables that are shared. A variable in $\overline{Op}$ is either a state variable, or an input or output variable. Since, all input and output variables are kept distinct, there are only two cases to consider: (i) State variables that are not shared by Op_1 and Op_2, and input and output variables; and (ii) State variables that are shared.

Case 1: The time-maps for unshared state variables in Op_1 and unshared state variables in Op_2 are separately constructed.

Case 1.1: All variables in Op_1 that are not shared by Op_2 (that includes state and, input and output variables) retain their maps from $\overline{Op_1}$.

$$M_1 = \{(s \mapsto \tau) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_1}) \wedge s \in \mathcal{D}(Op_1) \setminus \mathcal{D}(Op_2)\}$$

Case 1.2: The map for an unshared variable in Op_2 must be determined based on whether it is a state variable, an input variable or an output variable.

Case 1.2.1: A state variable that is not in the Δ-list of Op_2 is mapped to the invocation time of the operation Op_2; the timing parameter for a variable that is in the Δ-list of Op_2 should consider the pre and post-state of the operation. Combining these, the time-map for the variables in case 1.2.1 can be given as

$$\begin{aligned} M_2 = &\{(s \mapsto \overline{Op_2}.t) \mid s \in (\mathcal{D}(Op_2) \setminus (\mathcal{D}(Op_1) \cup \beta_?(Op_2) \cup \beta_!(Op_2) \cup \rho(Op_2))\} \cup \\ &\{(s \mapsto \overline{Op_2}.t) \mid s \in ((\mathcal{D}(Op_2) \setminus \mathcal{D}(Op_1)) \cap \rho(Op_2))\} \cup \\ &\{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid s \in ((\mathcal{D}(Op_2) \setminus \mathcal{D}(Op_1)) \cap \rho(Op_2))\} \end{aligned}$$

where $postVar(s)$ refers to the variable s in the postcondition (typically, it is identified as s' in the specification).

Case 1.2.2: The mapping for an input variable is given as

$$M_3 = \{(x \mapsto \overline{Op}.\epsilon_{1x}) \mid x \in \beta_?(Op_2)\}$$

Notice that $\overline{Op}.\epsilon_{1x} = \overline{Op_1}.\delta t + \overline{Op_2}.\epsilon_{1x}$

Case 1.2.3: The mapping for an output variable is given as

$$M_4 = \{(y \mapsto \overline{Op}.\epsilon_{2y}) \mid y \in \beta_!(Op_2)\}$$

Case 2: Only state variables are shared. There are four possible situations:

Case 2.1: If a shared variable is in the Δ-list of Op_1, but not in the Δ-list of Op_2, the mapping for that variable must be retained from $\Upsilon(\overline{Op_1})$.

Case 2.2: If a shared variable is not in the Δ-list of Op_1 and is not in the Δ-list of Op_2, the mapping for that variable must be copied as is from $\Upsilon(\overline{Op_1})$.

To include cases 2.1 and 2.2 in case 1.1, the definition of M_1 must be changed as

$$M_1 = \{(s \mapsto \tau) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_1}) \wedge (s \in \mathcal{D}(Op_1) \setminus \rho(Op_2))\}$$

Case 2.3: If a shared variable is in $\rho(Op_2)$, but not in $\rho(Op_1)$, the variable in the pre-state is mapped to $\overline{Op_2}.t$ and in the post-state it is mapped to $\overline{Op}.t + \overline{Op}.\delta t$.

Case 2.4: If a shared variable is in $\rho(Op_1)$ and in $\rho(Op_2)$, once again, its value is observable at the completion of Op and hence the variable is mapped to $\overline{Op}.t + \overline{Op}.\delta t$ after the operation is completed (i.e., in the post-state). However, the same variable in the pre-state is mapped to $\overline{Op}.t$. Map M_5 given below includes the situations described in cases 2.3 and 2.4:

$$\begin{aligned} M_5 = \; & \{(s \mapsto \overline{Op_2}.t) \mid s \in \rho(Op_2) \setminus \rho(Op_1)\} \cup \\ & \{(s \mapsto \overline{Op}.t) \mid s \in \rho(Op_2) \cap \rho(Op_1)\} \cup \\ & \{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid \\ & s \in \rho(Op_2) \setminus \rho(Op_1) \vee s \in \rho(Op_1) \cap \rho(Op_2)\} \end{aligned}$$

Based on this discussion, the formal semantics for sequential composition of filters is shown below:

Rule: (Sequential Composition)

$$\frac{Op = Op_1 \mathbin{⨟} Op_2 \vdash \qquad \overline{Op} = \overline{Op_1} \mathbin{⨟} \overline{Op_2} \vdash}{\begin{array}{l} \vdash \overline{Op}.t = \overline{Op_1}.t \wedge \overline{Op_2}.t = t + \overline{Op_1}.\delta t \wedge \overline{Op}.\delta t = \overline{Op_1}.\delta t + \overline{Op_2}.\delta t \\ x \in \beta_?(Op_1) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\epsilon_{1x} \\ y \in \beta_!(Op_1) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\epsilon_{2y} \\ x \in \beta_?(Op_2) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\delta t + \overline{Op_2}.\epsilon_{1x} \\ y \in \beta_!(Op_2) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\delta t + \overline{Op_2}.\epsilon_{2y} \\ \Upsilon(\overline{Op}) = M_1 \wedge M_2 \wedge M_3 \wedge M_4 \wedge M_5 \\ \Gamma_r(\overline{Op}) = \Gamma_r(\overline{Op_1}) \wedge \Gamma_r(\overline{Op_2}) \end{array}} \quad [\, p \,]$$

where $p \equiv \forall\, x \in \beta_?(Op_1) \cap \beta_?(Op_2) \bullet \overline{Op_2}[z/x] \wedge \overline{Op_2}[\epsilon_{1z}/\epsilon_{1x}] \wedge z \notin \beta_?(\overline{Op_1}) \wedge$
$\forall\, y \in \beta_!(Op_1) \cap \beta_!(Op_2) \bullet \overline{Op_2}[z/y] \wedge \overline{Op_2}[\epsilon_{2z}/\epsilon_{2y}] \wedge z \notin \beta_!(\overline{Op_1})$

The proviso ensures that all input and output variables are kept distinct, by renaming the common variables.

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 anticipated completion time of Op is the maximum of the anticipated completion times of Op_1 and Op_2. The input and output variables of Op are the union of the input and output variables of Op_1 and Op_2. The timing parameters of the input/output variables (the ϵ variables) for $\overline{Op}$ are copied from $\overline{Op_1}$ and $\overline{Op_2}$ without any modifications.

The time map of unshared state variables and those of the input and output variables do not change in the composite filter. However, the time map for the shared state variables deserve special attention. There are two cases to consider: (i) shared state variables that are in the Δ-list of either or both Op_1 and Op_2; and (ii) shared state variables that are in the Δ-list of neither Op_1 nor Op_2.

The time map of a shared state variable that is in the Δ-list of neither operations is retained from its respective operation. The time map of a shared state variable that is in the Δ-list of either Op_1 or Op_2 should be $\overline{Op}.t + \overline{Op}.\delta t$; the rationale is, regardless of the order and the number of times the variable is modified, its effect is observable only when the operation Op terminates.

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

$$\frac{Op = Op_1 \bigwedge Op_2 \vdash \qquad\qquad \overline{Op} = \overline{Op_1} \bigwedge \overline{Op_2} \vdash}{\begin{array}{l} \vdash \overline{Op}.t = \overline{Op_1}.t = \overline{Op_2}.t \\ \overline{Op}.\delta t = max(\overline{Op_1}.\delta t, \overline{Op_2}.\delta t) \\ x \in \beta_?(Op_1) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\epsilon_{1x} \\ x \in \beta_?(Op_2) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_2}.\epsilon_{1x} \\ y \in \beta_!(Op_1) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\epsilon_{2y} \\ y \in \beta_1(Op_2) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_2}.\epsilon_{2y} \\ \Upsilon(\overline{Op}) = \{(s \mapsto \tau) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_1}) \cup \Upsilon(\overline{Op_2}) \setminus (\Upsilon(\overline{Op_1}) \cap \Upsilon(\overline{Op_2}))\} \cup \\ \qquad\qquad \{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid s \in \rho(Op_1) \cup \rho(Op_2)\} \cup \\ \qquad\qquad \{(s \mapsto \overline{Op}.t) \mid s \in \rho(Op_1) \cup \rho(Op_2)\} \\ \Gamma_r(\overline{Op}) = \Gamma_r(\overline{Op_1}) \wedge \Gamma_r(\overline{Op_2}) \end{array}} \quad [\, p \,]$$

where $p \equiv \forall\, x \in \beta_?(Op_1) \cap \beta_?(Op_2) \bullet \overline{Op_2}[z/x] \wedge \overline{Op_2}[\epsilon_{1z}/\epsilon_{1x}] \wedge z \notin \beta_?(\overline{Op_1}) \wedge$
$\forall\, y \in \beta_!(Op_1) \cap \beta_!(Op_2) \bullet \overline{Op_2}[z/y] \wedge \overline{Op_2}[\epsilon_{2z}/\epsilon_{2y}] \wedge z \notin \beta_!(\overline{Op_1})$

Parallel Operator In Object-Z, the expression $Op_1 \parallel Op_2$ denotes a 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 because the timing parameters for the communicating variables (shared input and output variables) depend on which operation initiates the communication first. RTOZ resolves this problem by introducing the four parallel composition operators $Op_1 \overset{\parallel}{\leftarrow} Op_2$, $Op_1 \overset{\parallel}{\rightarrow} Op_2$, $Op_1 \overset{\parallel}{\leftrightarrows} Op_2$, $Op_1 \overset{\parallel}{\rightleftarrows} Op_2$, each 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.
- 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 several data exchanges between Op_1 and Op_2, but the notation makes it clear as to which operation acts first. The inference rules for the four operators are given next.

The invocation time $\overline{Op}.t$ for the composite operation $Op = Op_1 \overset{\parallel}{\leftarrow} Op_2$, is $\overline{Op_1}.t$ if Op_1 starts earlier than Op_2; otherwise, it is equal to $\overline{Op_2}.t$. The anticipated completion time for Op is the duration between $\overline{Op}.t$ and τ where τ is the time at which both Op_1 and Op_2 terminate. Notice that the operation Op_1 must be alive (must be invoked before and be ready to receive the input) when Op_2 sends it first output.

The input and output variables of the composite operations are the untion of input and output variables from Op_1 and Op_2 less the communicating input and output variables. The reason for eliminating the communicating input and output variables is that they are used only for data exchange between the two operations and therefore are not observable from Op. The derivation of offsets for input and output variables in Op is a bit tricky. Suppose, Op_1 starts earlier than Op_2. Let x be a non-communicating input variable of Op_1. Clearly, the timing variable ϵ_{1x} for Op is the same as ϵ_{1x} in Op_1 since Op starts at the same time as that of Op_1. However, if Op_2 starts earlier than Op_1, then ϵ_{1x} for Op is the sum of the ϵ_{1x} in Op_1 and the time difference between the invocation times of Op_1 and Op_2. Similar arguments apply when calculating the offsets for other non-communicating variables. The predicates *IOconstraints* given below describes the derivation of offsets for the non-communicating input and output variables.

<u>IOconstraints for Parallel Compositions:</u>

$$
\begin{aligned}
IOconstraints \equiv\ & x \in \beta_?(Op_1) \setminus \beta_!(Op_2) \Rightarrow \\
& \quad (\overline{Op}.t = \overline{Op_1}.t \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\epsilon_{1x}) \wedge \\
& \quad (\overline{Op}.t = \overline{Op_2}.t \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\epsilon_{1x} + (\overline{Op_1}.t - \overline{Op_2}.t)) \wedge \\
& x \in \beta_?(Op_2) \setminus \beta_!(Op_1) \Rightarrow \\
& \quad (\overline{Op}.t = \overline{Op_2}.t \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_2}.\epsilon_{1x}) \wedge \\
& \quad (\overline{Op}.t = \overline{Op_1}.t \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_2}.\epsilon_{1x} + (\overline{Op_2}.t - \overline{Op_1}.t)) \wedge \\
& y \in \beta_!(Op_1) \setminus \beta_?(Op_2) \Rightarrow \\
& \quad (\overline{Op}.t = \overline{Op_1}.t \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\epsilon_{2y}) \wedge \\
& \quad (\overline{Op}.t = \overline{Op_2}.t \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\epsilon_{2y} + (\overline{Op_1}.t - \overline{Op_2}.t)) \wedge \\
& y \in \beta_!(Op_2) \setminus \beta_?(Op_1) \Rightarrow \\
& \quad (\overline{Op}.t = \overline{Op_2}.t \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_2}.\epsilon_{2y}) \wedge \\
& \quad (\overline{Op}.t = \overline{Op_1}.t \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_2}.\epsilon_{2y} + (\overline{Op_2}.t - \overline{Op_1}.t))
\end{aligned}
$$

<u>Maplet constraints for Parallel Compositions:</u>

The time map for an unshared state variable that is not in the Δ-list of either operations is derived from $\Upsilon(\overline{Op_1})$ and $\Upsilon(\overline{Op_2})$. That is

$M_1 = \{(s \mapsto \tau) \mid (s \mapsto \tau) \in (\Upsilon(\overline{Op_1}) \cup \Upsilon(\overline{Op_2}) \setminus (\Upsilon(\overline{Op_1}) \cap \Upsilon(\overline{Op_2})) \wedge$
$\quad s \notin \rho(\overline{Op_1}) \cup \rho(\overline{Op_2})\}$

If a shared state variable is in the Δ-list of both the operations, the variable is mapped to $\overline{Op}.t$ in the pre-state, and it is mapped to $\overline{Op}.t + \overline{Op}.\delta t$ in the post-state. That is,

$M_2 = \{(s \mapsto \overline{Op}.t) \mid s \in \rho(\overline{Op_1}) \cap \rho(\overline{op_2})\} \cup$
$\quad \{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid s \in \rho(\overline{Op_1}) \cap \rho(\overline{Op_2})\}$

If a state variable belongs to the Δ-list of only one of the component operations, its mapping in the pre-state is unchanged, and the variable is mapped to $\overline{Op}.\delta t$ in the post-state. This map is,

$M_3 = \{(s \mapsto \tau) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_1}) \wedge s \in \rho(\overline{Op_1}) \wedge s \notin \rho(\overline{Op_2}))\} \cup$
$\quad \{(s \mapsto \tau) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_2}) \wedge s \in \rho(\overline{Op_2}) \wedge s \notin \rho(\overline{Op_1}))\}$

and

$M_4 = \{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_1}) \wedge s \in \rho(\overline{Op_1}) \wedge s \notin \rho(\overline{Op_2}))\} \cup$
$\quad \{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_2}) \wedge s \in \rho(\overline{Op_2}) \wedge s \notin \rho(\overline{Op_1}))\}$

Finally, the maplet constraints for parallel composition operators can be summarized as

$MapletConstraints = M_1 \wedge M_2 \wedge M_3 \wedge M_4$

Proviso for Parallel Compositions:

$p \equiv \forall\, x \in \beta_?(Op_1) \cap \beta_?(Op_2) \bullet \overline{Op_2}[z/x] \wedge \overline{Op_2}[\epsilon_{1z}/\epsilon_{1x}] \wedge z \notin \beta_?(\overline{Op_1}) \wedge$
$\quad \forall\, y \in \beta_!(Op_1) \cap \beta_!(Op_2) \bullet \overline{Op_2}[z/y] \wedge \overline{Op_2}[\epsilon_{2z}/\epsilon_{2y}] \wedge z \notin \beta_!(\overline{Op_1})$

The formal semantics for parallel left composition is given below:

Rule: (Parallel Left)

$$\frac{Op = Op_1 \overset{\parallel}{\leftarrow} Op_2 \vdash \qquad \overline{Op} = \overline{Op_1} \overset{\parallel}{\leftarrow} \overline{Op_2} \vdash}{\begin{array}{l}\vdash \overline{Op}.t = min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \overline{Op_1}.t \leq \overline{Op_2}.t + \underset{y \in \beta_!(Op_2)}{min}(\overline{Op_2}.\epsilon_{2y}) \wedge \\ \overline{Op}.\delta t = max(\overline{Op_1}.t + \overline{Op_1}.\delta t, \overline{Op_2}.t + \overline{Op_2}.\delta t) - min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \\ IOconstraints \wedge MapletConstraints\end{array}} \quad [\, p \,]$$

The formal definitions for the other three parallel filter operators are similar except for the invocation times of the two operations Op_1 and Op_2. These are given below:

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

$$\frac{Op = Op_1 \overset{\parallel}{\rightarrow} Op_2 \vdash \qquad \overline{Op} = \overline{Op_1} \overset{\parallel}{\rightarrow} \overline{Op_2} \vdash}{\begin{array}{l}\vdash \overline{Op}.t = min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \overline{Op_2}.t \leq \overline{Op_1}.t + \underset{y \in \beta_!(Op_1)}{min}(\overline{Op_1}.\epsilon_{2y}) \wedge \\ \overline{Op}.\delta t = max(\overline{Op_1}.t + \overline{Op_1}.\delta t, \overline{Op_2}.t + \overline{Op_2}.\delta t) - min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \\ IOconstraints \wedge MapletConstraints\end{array}} [\ p\ ]$$

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

$$\frac{Op = Op_1 \overset{\parallel}{\leftrightarrows} Op_2 \vdash \qquad \overline{Op} = \overline{Op_1} \overset{\parallel}{\leftrightarrows} \overline{Op_2} \vdash}{\begin{array}{l}\vdash \overline{Op}.t = min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \\ \overline{Op_1}.t \leq \overline{Op_2}.t + \underset{y \in \beta_!(Op_2)}{min}(\overline{Op_2}.\epsilon_{2y}) \wedge \\ \overline{Op_2}.t + \overline{Op_2}.\delta t \geq \overline{Op_1}.t + \underset{y \in \beta_!(Op_1)}{max}(\overline{Op_1}.\epsilon_{2y}) \wedge \\ \overline{Op}.\delta t = max(\overline{Op_1}.t + \overline{Op_1}.\delta t, \overline{Op_2}.t + \overline{Op_2}.\delta t) - min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \\ IOconstraints \wedge MapletConstraints\end{array}} [\ p\ ]$$

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

$$\frac{Op = Op_1 \overset{\parallel}{\rightleftarrows} Op_2 \vdash \qquad \overline{Op} = \overline{Op_1} \overset{\parallel}{\rightleftarrows} \overline{Op_2} \vdash}{\begin{array}{l}\vdash \overline{Op}.t = min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \\ \overline{Op_2}.t \leq \overline{Op_1}.t + \underset{y \in \beta_!(Op_1)}{min}(\overline{Op_1}.\epsilon_{2y}) \wedge \\ \overline{Op_1}.t + \overline{Op_1}.\delta t \geq \overline{Op_2}.t + \underset{y \in \beta_!(Op_2)}{max}(\overline{Op_2}.\epsilon_{2y}) \wedge \\ \overline{Op}.\delta t = max(\overline{Op_1}.t + \overline{Op_1}.\delta t, \overline{Op_2}.t + \overline{Op_2}.\delta t) - min(\overline{Op_1}.t, \overline{Op_2}.t) \wedge \\ IOconstraints \wedge MapletConstraints\end{array}} [\ p\ ]$$

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. The invocation time of Op is the same as the invocation time of the operation that is selected. The completion time of Op is the same as that of the invoked operation. The input and output variables of Op are the input and output variables of the operation that is invoked.

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

$$\frac{Op = Op_1 \; [] \; Op_2 \vdash \qquad \overline{Op} = \overline{Op_1} \; [] \; \overline{Op_2} \vdash}{\begin{array}{l} \vdash (Op_1.t = Op.t \wedge Op.\delta t = Op_1.\delta t \wedge \\ x \in \beta_?(Op_1) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\epsilon_{1x} \wedge \\ y \in \beta_!(Op_1) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\epsilon_{2y} \wedge \\ \mathcal{P}(\overline{Op}) = \mathcal{P}(\overline{Op_1})) \\ \boxplus \\ (Op_2.t = Op.t \wedge Op.\delta t = Op_2.\delta t \wedge \\ x \in \beta_?(Op_2) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_2}.\epsilon_{1x} \wedge \\ y \in \beta_!(Op_2) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_2}.\epsilon_{2y} \wedge \\ \mathcal{P}(\overline{Op}) = \mathcal{P}(\overline{Op_2})) \end{array}}$$

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

Environment Enrichment Let $Op = Op_1 \bullet Op_2$. In this case, Op_2 starts immediately after Op_1 terminates as in sequential composition, but Op_2 has access to the signature of Op_1. Correspondingly, the filter $\overline{Op_2}$ has access to the signature of the filter $\overline{Op_1}$. The time map of $\overline{Op_1}$ will be retained in $\overline{Op}$. The time map for the variables in the signature of $\overline{Op_2}$ will be calculated according to the following rules:

A state variable that is not in the Δ-list of Op_2 will be mapped to $\overline{Op_2}.t$ in the pre-state and is mapped to $\overline{Op_2}.t + \overline{Op_2}.\delta t$ in the post-state. That is,

$M_1 = \{(s \mapsto \overline{Op_2}.t) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_2}) \wedge s \in (\alpha_1(OP_2) \cup \alpha_2(Op_2) \setminus \rho(Op_2))\}$ for some τ

and

$M_2 = \{(postVar(s) \mapsto \overline{Op_2}.t + \overline{Op_2}.\delta t) \mid$
$(s \mapsto \tau) \in \Upsilon(\overline{Op_2}) \wedge s \in (\alpha_1(Op_2) \cup \alpha_2(Op_2) \setminus \rho(Op_2))\}$ for some τ

The time map of input/output variables in Op_2 will be updated such that the term $\overline{Op_1}.t + \overline{Op_1}.\delta t$ is added to the ϵ variables in Op_2. That is,

$M_3 = \{(x \mapsto \overline{Op}.\epsilon_{1x}) \mid x \in (\beta_?(\overline{Op_1}) \cup \beta_?(\overline{Op_2})\} \cup$
$\{(y \mapsto \overline{Op}.\epsilon_{2y}) \mid y \in \beta_!(\overline{Op_1}) \cup \beta_!(\overline{Op_2})\}$

The time-map for a variable in the Δ-list of Op_1 and Op_2 should be overwritten by the maplet $\{postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t\}$ in $\Upsilon(\overline{Op_1})$. Thus,

$M_4 = \Upsilon(\overline{Op_1}) \oplus \{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid s \in \rho(\overline{Op_1}) \cap \rho(\overline{Op_2})\}$

A state variable that is in the Δ-list of Op_2 but not in the Δ-list of Op_1 is mapped to $\overline{Op_2}.t$ in the pre-state and it is mapped to $\overline{Op}.t + \overline{Op}.\delta t$ in the post-state. This is formally defined as

$M_5 = \{(s \mapsto \overline{OP_2}.t) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_2}) \wedge s \in \rho(Op_2) \wedge s \notin \rho(Op_1)\} \cup$
$\{(postVar(s) \mapsto \overline{Op}.t + \overline{Op}.\delta t) \mid (s \mapsto \tau) \in \Upsilon(\overline{Op_2}) \wedge s \in \rho(Op_2) \wedge s \notin \rho(Op_1)\}$

Rule: (Enrichment)

$$\frac{Op = Op_1 \bullet Op_2 \vdash \qquad \overline{Op} = \overline{Op_1} \bullet \overline{Op_2} \vdash}{\begin{array}{l} \vdash \overline{Op}.t = \overline{Op_1}.t \wedge \overline{Op_2}.t = \overline{Op_1}.t + \overline{Op_1}.\delta t \\ \overline{Op}.\delta t = \overline{Op_1}.\delta t + \overline{Op_2}.\delta t \\ x \in \beta_?(Op_1) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\epsilon_{1x} \\ y \in \beta_!(Op_1) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\epsilon_{2y} \\ x \in \beta_?(Op_2) \Rightarrow \overline{Op}.\epsilon_{1x} = \overline{Op_1}.\delta t + \overline{Op_2}.\epsilon_{1x} \\ y \in \beta_!(Op_2) \Rightarrow \overline{Op}.\epsilon_{2y} = \overline{Op_1}.\delta t + \overline{Op_2}.\epsilon_{2y} \\ \Upsilon(\overline{Op}) = \Upsilon(\overline{Op_1}) \cup M_1 \cup M_2 \cup M_3 \cup M_4 \cup M_5 \\ \Gamma_r(\overline{Op}) = \Gamma_r(\overline{Op_1}) \wedge \Gamma_r(\overline{Op_2}) \end{array}} \quad [\ p\ ]$$

where $p \equiv \forall\, x \in \beta_?(Op_1) \cap \beta_?(Op_2) \bullet \overline{Op_2}[z/x] \wedge \overline{Op_2}[\epsilon_{1z}/\epsilon_{1x}] \wedge z \notin \beta_?(\overline{Op_1}) \wedge$
$\forall\, y \in \beta_!(Op_1) \cap \beta_!(Op_2) \bullet \overline{Op_2}[z/y] \wedge \overline{Op_2}[\epsilon_{2z}/\epsilon_{2y}] \wedge z \notin \beta_!(\overline{Op_1})$

4.4 Semantics for Filter 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, the result in this case is a conjunction of the two operations; i.e., $Op_{effective} \mathrel{\widehat=} Op_{sub} \wedge Op_{sup}$. Object-Z imposes the restriction that Op_{sub} cannot contradict Op_{sup}; should this happen, one of the operations must be renamed.

RTOZ extends the inheritance mechanism of Object-Z to filter specifications. 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 $\overline{Op}$ is redefined in C_{sub}_**filter**, the timing predicate of $\overline{Op}$ in C_{sub}_**filter** is conjoined with the timing predicate $\overline{Op}$ in C_{sup}_**filter**; the **timemap** of $\overline{Op}$ in C_{sub}_**filter** overrides the **timemap** of $\overline{Op}$ in C_{sup}. In other words, there will be only one consistent **timemap** in the result after inheritance. It is easy to observe that C_{sup}_**filter** $\subset$ C_{sub}_**filter** because a subclass must necessarily be different from its superclass. This necessitates that the filter specification corresponding to a subclass must 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** must be renamed by w in C_{sub}_**filter**. That is,
  Op *in* C_{sub}_**filter** $\mathrel{\widehat=}$ $Op[w/v]$ *in* C_{sup}_**filter**

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

- An additional state variable v is introduced in C_{sub}. Let $Op_1, Op_2, \ldots, Op_n$, $n \geq 0$ be the operations in C_{sup} which are modified due to the addition of v. Because of incremental specification, each of the operations Op_1, Op_2, $\ldots$, Op_n in C_{sub} will have additional constraints. Consequently, there will be new filters $\overline{Op_1}$, $\overline{Op_2}$, $\ldots$, $\overline{Op_n}$ in C_{sub}**_filter**. Each of these filters will include additional mappings and may have additional constraints due to the introduction of the variable v. Formally,

$$\overline{Op_i}\ in\ C_{sub}\_\mathbf{filter} \supset \overline{Op_i}\ in\ C_{sup}\_\mathbf{filter}, \qquad 1 \leq i \leq n$$

- An additional operation Op_{new} is introduced in C_{sub}. Consequently, there will be a filter $\overline{Op_{new}}$ in C_{sub}**_filter**.
- An operation Op in C_{sup} is redefined in C_{sub}. In this case, there will be two versions of Op and two versions of the filter $\overline{Op}$. According to the semantics of Object-Z, the two versions of the operation Op will be conjoined together to form a single operation OP in C_{sub}. In RTOZ, the corresponding filters are also conjoined to have one single filter $\overline{OP}$ in the class C_{sub}**_filter**.
- In RTOZ, one can introduce an additional filter $\overline{Op_{add}}$ in C_{sub}**_filter** for an operation Op in the superclass, even when the subclass does not redefine the operation. In this situation, the filter $\overline{Op_{add}}$ will replace the filter $\overline{Op}$ in the subclass, provided that $\mathcal{P}\ (\overline{Op_{add}}) \Rightarrow \mathcal{P}\ (\overline{Op})$. This condition implies that the new filter must preserve the timing behavior of the old filter.

The inference rule for filter inheritance given below uses the same convention as the inference rule for class inheritance given in [18]. Accordingly, the premises include the set of filter specifications $C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_n$. 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 the filter specifications $C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_{n-1}$. Thus, the filter $\overline{Op}$ in $C\_\mathbf{filter}_n$ will be derived from each of these filter specifications. A formal definition of the rule for inheritance follows:
<u>Rule:</u> (FilterInheritance)

$$\frac{C\_\mathbf{filter}_1 ::\mid Op_1 \in \Omega(C\_\mathbf{filter}_1) \vdash \qquad \ldots \qquad C\_\mathbf{filter}_n ::\mid Op_n \in \Omega(C\_\mathbf{filter}_n) \vdash}{\begin{array}{l} C\_\mathbf{filter} ::\mid Op \in \Omega(C\_\mathbf{filter}) \vdash\uparrow Op = \big[\mathcal{D}(Op_1) \cup \ldots \cup \mathcal{D}(Op_n) \mid \\ \quad \Upsilon(Op) = \Upsilon(Op_n) \oplus \ldots \oplus \Upsilon(Op_1) \wedge \\ \quad \Gamma_r(Op) = \Gamma_r(Op_1) \wedge \ldots \wedge \Gamma_r(Op_n)\big] \end{array}} \quad [\,p\,]$$

where $p \equiv \omega(\uparrow Op) = \omega(Op_1) = \ldots = \omega(Op_n)$

$$\begin{array}{l} \iota(C\_\mathbf{filter}) = \{C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_n\} \\ \phi(Op_1) \subseteq \alpha(C\_\mathbf{filter}_1) \\ \vdots \\ \phi(Op_n) \subseteq \alpha(C\_\mathbf{filter}_n) \end{array}$$

4.5 Semantics for Polymorphic Substitution of Filters

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 an operation Op, two cases arise: (1) If Op is defined in C_{sup} and is not redefined in C_{sub}, then Op in C_{sup} will respond to the invocation; and (2) If Op is defined in C_{sup} and is redefined in C_{sub}, then the logical conjunction of $C_{sup}.Op$ and $C_{sub}.Op$ becomes the response to the invocation. Case 1 will not pose any problems in RTOZ. However, Case 2 deserves special attention. The condition for universal polymorphism is that if an object O_{sub} belonging to C_{sub} is substituted for an object O_{sup} belonging to C_{sup}, then O_{sub} should behave identically to O_{sup} in all respects. That is, the filter specification C_{sub}**_filter** should be more restrictive than the filter specification C_{sup}**_filter**. In other words, the timing requirements of any operation invoked by O_{sub} must satisfy the timing requirements defined within the filter specification for O_{sup}. Since Op in C_{sub} can only define additional constraints that are consistent with the set of constraints already defined for Op in C_{sup}, the timing requirements in filter $\overline{Op}$ in C_{sub}**_filter** can only include additional maplets and/or additional timing constraints. This leads to the following semantics for polymorphic substitution:

- For every filter $\overline{Op}$ in C_{sub}**_filter**, the **timemap** of $\overline{Op}$ must be a superset of the **timemap** of the filter $\overline{Op}$ in C_{sup}**_filter** and the timing constraints in the former must imply the timing constraints in the latter.

The inference rule describing the semantics for polymorphic substitution is given below. In this rule, the term a denotes a polymorphic object belonging to any of the subclasses of a given class.

Rule: (Polymorphism)

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

where $p \equiv (\iota^{\sim})^{*}(\!|\ C\ |\!) = \{C_1, \ldots, C_n\}$
$\qquad (\iota^{\sim})^{*}(\!|\ C\_\mathbf{filter}\ |\!) = \{C\_\mathbf{filter}_1, \ldots, C\_\mathbf{filter}_n\}$

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. The object a belongs to only one of the descendent classes of C.

5 Case Study

This section describes an RTOZ specification of a simplified electronic mail system. Although simple, the specification uses all the composite operators in Object-Z.

5.1 Problem Description and Informal Design

The electronic mail system under consideration consists of a network of nodes. A network manager handles communication of mails between the nodes. A node wishing to send a mail (hereafter referred to as *sender*) composes the mail first and then submits it to the network manager. The sender will get an acknowledgement from the network manager only if the mail is accepted by the latter. The descision to accept or reject a mail is based on the status of the network manager's buffer to hold the mail. The network manager should deliver every accepted mail within a prescribed time to its addressee. It is also responsible for informing the sender of an accepted mail whether or not the mail was delivered within the prescribed time. The following are the assumptions made during modeling:

- If a sender receives a notification from the network manager that a mail is not accepted, the sender can decide to send the mail again or to ignore the mail. This portion of the sender's activity is not included in the specification.
- Each mail is addressed to only one recipient. Therefore, if a sender wants to send a mail to more than one recipient, the sender must send a separate mail to each recipient.
- The internal charactersitics of the communication channels are not modeled. Consequently, time constraints that are related to the activities of the communication channels such as storing, sorting, checking space availability are ignored.

The following timing constraints should be met:

- When a sender submits a mail to the network, the sender must receive an acknowledgement from the network manager within t_{ack} time units. If an acknowledgement is not received within t_{ack} time units, the sender will assume that the mail is lost.
- A mail that is accepted by the network must be delivered within $t_{deliver}$ time units; if not, the sender will be notified and the mail will be deleted from the network.
- The maximum time allowed to transfer a mail from the network manager's buffer to a receiver's input buffer is $t_{dispatch}$.

Informal Design When a mail is accepted by the network manager, the latter puts a timestamp ts on the mail which indicates the time of acceptance of the mail. The mail should be delivered to the recipient within $t_{deliver}$ time units after it has been accepted by the network manager. If the mail is delivered within $t_{deliver}$ time units, the network manager will send a notification to the sender at the time when the mail is delivered. If the network manager is unable to deliver the mail within $t_{deliver}$ time units, the former sends a notification to the sender informing that the mail has not been delivered and also deletes the mail from the network manager's buffer.

The network manager maintains two queues for holding mails - *regular queue* which is used to hold the incoming mails, and *retry queue* which is used to hold the mails that are not dispatched in the first attempt. The network manager concurrently dispatches the mails from both the queues. Each node in the network has an incoming tray in which the mails addressed to that node are placed by the network manager. If the incoming tray is full, it is interpreted that the network manager fails to place the mail in the incoming tray. In this case, the network manager will decide either to place the mail in the *retry_queue* if the $t_{deliver}$ time limit for the mail has not expired, or it will delete the mail and inform the sender. Receiving a mail at a node is different from reading the mail. While receiving a mail, the network manager attempts to put the mail in the incoming tray of the receiver. When a node reads a mail, the node transfers the mail from its incoming tray to an internal mailbox.

5.2 RTOZ Specification - First Tier

The following basic types are introduced in the specification; these are globally accessible by all class definitions:

$$[NODEID, CHAR]$$

NODEID refers to the set of node identifiers used in the specification, and *CHAR* represents the character type similar to that defined in programming languages. A message is modeled as a sequence of characters. The following global types are used.

$$\begin{aligned}
&MailContent == \mathrm{seq}\ CHAR \\
&Acknowledgement ::= Accepted \mid NotAccepted \\
&DeliveryMessage ::= Delivered \mid TimeOut \\
&SpaceAvailability ::= Full \mid NotFull
\end{aligned}$$

The type *MailContent* represents the message portion of a mail. The free type *Acknowledgement* is used by the network manager to send an acknowledgement to a sender indicating whether or not the mail submitted by the sender has been accepted. The network manager uses the type *DeliveryMessage* to indicate whether or not a mail is delivered to its recipient. *SpaceAvailability* is a free type that is used to check whether or not a given buffer is full.

Mail Class The class *Mail* contains the following structural components: *from* identifies the sender, *to* identifies the recipient, *body* denotes the content of the mail, and *timestamp* indicates the time at which the mail is accepted by the network manager. The timestamp field is used only by the network manager. The state invariant of *Mail* asserts that the body of the mail should not be empty. The subject field of a mail is included in the body itself. The *Mail* class contains three operations - one operation to edit the address of the recipient, another operation to edit the body of the mail, and a third operation to change the timestamp of the mail. All the three operations are exported.

Mail

$from : NODEID$
$to : NODEID$
$body : MailContent$
$timestamp : RealTime$

$body \neq \langle\rangle$

EditTo

$\Delta(to)$

$to' \neq to$

EditBody

$\Delta(body)$

$body' \neq body$

StampIt

$\Delta(timestamp)$
$ts? : RealTime$

$timestamp' = ts?$

MailQueue Class The class *MailQueue* defines a buffer for holding mails. There are three mail buffers in the applications; these three buffers are the *incoming tray* in a node, and the *regular queue* and the *retry queue* both owned by the network manager. All of them are modeled using the class *MailQueue*. The buffer is defined as an injective sequence of *Mail*. Hence, there are no duplications in the queue[§]. The size of the buffer is limited and is denoted by the constant $buffer\_capacity$. Initially, the buffer is empty (denoted by the $INIT$ state). The following operations are defined in the class *MailQueue*:

AddMail When a mail is added to the queue, it is always placed at the end of the queue. The precondition for the operation asserts whether or not the buffer is full.

ExtractMail This operation is invoked to extract the mail from the front of the queue. The precondition ensures that the buffer is non-empty. The extracted mail is returned as *out!*.

IsFull This operation returns a message indicating whether or not the queue is full.

[§]Two instances of class *Mail* are equal if and only if the respective fields in their state spaces are equal.

$$
\begin{array}{|l}
\hline
\textit{MailQueue} \\
\quad buffer\_capacity : \mathbb{N}_1 \\
\quad \boxed{\begin{array}{l} buffer : \mathrm{iseq}\, Mail \\ \hline \#buffer \leq buffer\_capacity \end{array}} \\
\quad \textit{INIT}: \boxed{buffer = \langle\rangle} \\
\quad \textit{AddMail} \\
\quad \boxed{\begin{array}{l} \Delta(buffer) \\ new? : Mail \\ \hline \#buffer < buffer\_capacity \\ buffer' = buffer \frown \langle new? \rangle \end{array}} \\
\quad \textit{ExtractMail} \\
\quad \boxed{\begin{array}{l} \Delta(buffer) \\ out! : Mail \\ \hline \#buffer > 0 \\ buffer' = tail\ buffer \\ out! = head\ buffer \end{array}} \\
\quad \textit{IsFull} \\
\quad \boxed{\begin{array}{l} result! : SpaceAvailability \\ \hline \#buffer = buffer\_capacity \Rightarrow result! = Full \\ \#buffer \neq buffer\_capacity \Rightarrow result! = NotFull \end{array}} \\
\hline
\end{array}
$$

Node Class The class *Node* has three structural components: nid indicates the unique identification number of the node, $intray$ represents the buffer to hold the incoming mails, and $mbox$ represents the buffer to hold the mails that are read by this node. The mails in $mbox$ are categorized by the senders' identifications. Therefore, $mbox$ is defined as a partial function from node identifiers to a set of mails, indicating the senders and the set of mails received from each sender. The state invariant of the class *Node* asserts that (i) *mbox* of the current node has an entry for a node n if the current node has received at least one mail from n; (ii) the current node is indeed the receiver of each mail received by the node; (iii) all the mails are classified according to the respective senders' identifications; and (iv) the mails in the incoming tray and the mails in the mail box are distinct; i.e. no mail appears both in the incoming tray and in the mailbox at the same time.

Node

$$
\begin{array}{l}
nid : NODEID \\
intray : MailQueue \\
mbox : NODEID \pfun \mathbb{P}\, Mail \\
\hline
\forall\, id : NODEID \mid id \in \operatorname{dom} mbox \bullet \\
\quad (mbox(id) \neq \varnothing) \land \\
\quad (\forall\, m : Mail \mid m \in mbox(id) \bullet m.from = id \land nid = m.to) \land \\
\quad (mbox(id) \cap \operatorname{ran}(intray.buffer) = \varnothing)
\end{array}
$$

Send

$$
\begin{array}{l}
to? : NODEID \\
body? : MailContent \\
m! : Mail \\
\hline
m!.from = nid \\
m!.to = to? \\
m!.body = body?
\end{array}
$$

ReceiveACK

$$
ack? : Acknowledgement
$$

ReceiveDeliveryMessage

$$
delivery? : DeliveryMessage
$$

$$
SendAndWait \mathrel{\widehat{=}} Send \fatsemi ReceiveACK
$$

$$
Receive \mathrel{\widehat{=}} [\, new? : Mail \mid nid = new?.to \,] \bullet intray.AddMail
$$

UpdateMailbox

$$
\begin{array}{l}
\Delta(mbox) \\
m? : Mail \\
\hline
(m?.from \in \operatorname{dom} mbox \Rightarrow \\
\quad mbox' = mbox \oplus \{m?.from \mapsto mbox(m?.from) \cup \{m?\}\}) \\
(m?.from \notin \operatorname{dom} mbox \Rightarrow \\
\quad mbox' = mbox \oplus \{m?.from \mapsto \{m?\}\})
\end{array}
$$

$$
Read \mathrel{\widehat{=}} intray.ExtractMail \bullet UpdateMailbox[out!/m?]
$$

$$
\begin{array}{l}
Forward \mathrel{\widehat{=}} [\, m?, m! : Mail \,] \bullet \\
\quad ((Read \fatsemi m?.EditTo \fatsemi SendAndWait) \bullet [\, m?.from \neq m!.to \,])
\end{array}
$$

$$
\begin{array}{l}
Reply \mathrel{\widehat{=}} [\, m?, m! : Mail \,] \bullet \\
\quad ((Read \fatsemi m?.EditBody \fatsemi SendAndWait) \bullet [\, m?.from = m!.to \,])
\end{array}
$$

The operations defined in the class *Node* have the following significance:

Send This operation is invoked by a node when it wants to send a mail. The address of the recipient and the body of the mail are both required to send a mail.

ReceiveACK A node invokes *ReceiveACK* to receive an acknowledgement from the network manager after the node has submitted a mail. The actions of the node after receiving the acknowledgement are not relevant to the current model and hence are not included in the specification.

ReceiveDeliveryMessage This operation is invoked by a node to receive the delivery notification from the network manager. The actions of the node after receiving the delivery notification is not important for the current discussion and hence are not included in the specification.

SendAndWait This is a composite operation invoked by a node when it wants to send a mail and to wait for an acknowledgement from the network manager.

Receive This operation is invoked by a node to receive a mail from the network manager. As a consequence of this operation, the mail will be added to the incoming tray of the node. It is important to notice that receiving a mail does not imply reading the mail; the node will read the mail later using the *Read* operation (to be discussed shortly).

UpdateMailbox When a node n reads a mail m from its incoming tray sent by a node p, the node n will place m along with the mails received from p before, or will create a new entry for p in *mbox* in n and add m to that entry.

Read Reading a mail is a composite operation that consists of two tasks: extracting the mail from the incoming tray and updating the mailbox.

Forward A node can forward a mail that it has read by editing only the *to* field of the mail and sending it to a new address.

Reply A node can reply to a previously read mail by editing only the body of the mail, and sending it back to the sender of the original mail.

Network Manager Class The class *Network Manager* consists of three structural components - *nodes* identifies the set of node identifiers in the network, *regular_queue* holds the newly arriving mails, and *retry_queue* holds the mails that are not delivered in the first attempt. The state invariant of this class asserts that (i) no mail sits in both the queues at the same time; and (ii) the sender and receiver of each mail in each queue is known to the network manager. Initially, both the queues are empty.

NetworkManager

$$
\begin{array}{l}
nodes : \mathbb{P}\, NODEID \\
regular\_queue : MailQueue \\
retry\_queue : MailQueue \\
\hline
\mathrm{ran}(regular\_queue.buffer) \cap \mathrm{ran}(retry\_queue.buffer) = \varnothing \\
\forall\, m : Mail \mid m \in \mathrm{ran}(regular\_queue.buffer) \cup \mathrm{ran}(retry\_queue.buffer) \bullet \\
\quad m.from \in nodes \land m.to \in nodes
\end{array}
$$

INIT

$$
\begin{array}{l}
regular\_queue.buffer = \varnothing \\
retry\_queue.buffer = \varnothing
\end{array}
$$

$$
\begin{array}{l}
AcceptMail \mathrel{\widehat=} \\
\quad [\, ack! : Acknowledgement;\ result! : SpaceAvailability;\ m! : Mail; \\
\qquad sender : Node \mid sender.nid \in nodes \,] \bullet \\
\quad (sender.SendAndWait) \overset{\parallel}{\rightleftarrows} \\
\qquad ((regular\_queue.IsFull \land [\, result! = NotFull \land ack! = Accepted \,]) \mathbin{⨾} \\
\qquad (regular\_queue.AddMail[m!/new?] \land m!.StampIt)) \\
RefuseMail \mathrel{\widehat=} \\
\quad [\, ack! : Acknowledgement;\ result! : SpaceAvailability; \\
\qquad sender : Node \mid sender.nid \in nodes \,] \bullet \\
\quad (sender.SendAndWait) \overset{\parallel}{\rightleftarrows} \\
\qquad (regular\_queue.IsFull \land [\, result! = Full \land ack! = NotAccepted \,]) \\
Regular\_dispatch\_successful \mathrel{\widehat=} \\
\quad [\, receiver : Node;\ tray : MailQueue;\ out! : Mail;\ result! : SpaceAvailability \mid \\
\qquad receiver.nid \in nodes \land out!.to = receiver.nid \land tray = receiver.intray \,] \bullet \\
\qquad ((tray.IsFull \land [\, result! = NotFull \,]) \mathbin{⨾} \\
\qquad\quad (regular\_queue.ExtractMail \overset{\parallel}{\rightarrow} receiver.Receive[out!/new?])) \\
Retry\_dispatch\_successful \mathrel{\widehat=} \\
\quad [\, receiver : Node;\ tray : MailQueue;\ out! : Mail;\ result! : SpaceAvailability \mid \\
\qquad receiver.nid \in nodes \land out!.to = receiver.nid \land tray = receiver.intray \,] \bullet \\
\qquad ((tray.IsFull \land [\, result! = NotFull \,]) \mathbin{⨾} \\
\qquad\quad (retry\_queue.ExtractMail \overset{\parallel}{\rightarrow} receiver.Receive[out!/new?])) \\
Regular\_dispatch\_fail \mathrel{\widehat=} \\
\quad [\, receiver : Node;\ tray : MailQueue;\ out! : Mail;\ result! : SpaceAvailability \mid \\
\qquad receiver.nid \in nodes \land out!.to = receiver.nid \land tray = receiver.intray \,] \bullet \\
\qquad ((tray.IsFull \land [\, result! = Full \,]) \mathbin{⨾} \\
\qquad\quad (regular\_queue.ExtractMail \overset{\parallel}{\rightarrow} retry\_queue.AddMail[out!/new?]))
\end{array}
$$

$$
\begin{array}{l}
\textit{NetworkManager}(\textit{contd}\ldots) \\
\hline
\textit{Retry\_dispatch\_fail} \mathrel{\widehat=} \\
\quad \big[\, \textit{receiver} : \textit{Node};\ \textit{tray} : \textit{MailQueue};\ \textit{out}! : \textit{Mail};\ \textit{result}! : \textit{SpaceAvailability} \,\big| \\
\qquad \textit{receiver}.\textit{nid} \in \textit{nodes} \land \textit{out}!.\textit{to} = \textit{receiver}.\textit{nid} \land \textit{tray} = \textit{receiver}.\textit{intray} \,\big] \bullet \\
\qquad ((\textit{tray}.\textit{IsFull} \land [\, \textit{result}! = \textit{Full} \,]) \mathbin{\S} \\
\qquad\quad (\textit{retry\_queue}.\textit{ExtractMail} \stackrel{\parallel}{\rightarrow} \textit{retry\_queue}.\textit{AddMail}[\textit{out}!/\textit{new}?])) \\
\textit{Delivered} \mathrel{\widehat=} \\
\quad \big[\, \textit{sender} : \textit{Node};\ \textit{out}! : \textit{Mail};\ \textit{delivery}! : \textit{DeliveryMessage} \,\big| \\
\qquad \textit{sender}.\textit{nid} \in \textit{nodes} \land \textit{out}!.\textit{from} = \textit{sender}.\textit{nid} \,\big] \bullet \\
\quad ([\, \textit{delivery}! = \textit{Delivered} \,] \stackrel{\parallel}{\rightarrow} \textit{sender}.\textit{ReceiveDeliveryMessage}) \\
\textit{NotDelivered} \mathrel{\widehat=} \\
\quad \big[\, \textit{sender} : \textit{Node};\ \textit{out}! : \textit{Mail};\ \textit{delivery}! : \textit{DeliveryMessage} \,\big| \\
\qquad \textit{sender}.\textit{nid} \in \textit{nodes} \land \textit{out}!.\textit{from} = \textit{sender}.\textit{nid} \,\big] \bullet \\
\quad ([\, \textit{delivery}! = \textit{TimeOut} \,] \stackrel{\parallel}{\rightarrow} \textit{sender}.\textit{ReceiveDeliveryMessage}) \\
\textit{Regular\_dispatch} \mathrel{\widehat=} \\
\quad (\textit{Regular\_dispatch\_successful} \mathbin{\S} \textit{Delivered})\ [] \\
\quad ([\, \textit{delivery}! : \textit{DeliveryMessage} \,] \bullet \textit{Regular\_dispatch\_fail}) \\
\textit{Retry\_dispatch} \mathrel{\widehat=} \\
\quad (\textit{Retry\_dispatch\_successful} \mathbin{\S} \textit{Delivered})\ [] \\
\quad ([\, \textit{delivery}! : \textit{DeliveryMessage} \,] \bullet \textit{Retry\_dispatch\_fail})\ [] \\
\quad ([\, \textit{result}! : \textit{SpaceAvailability} \,] \bullet \textit{NotDelivered}) \\
\textit{Dispatch} \mathrel{\widehat=} \textit{Regular\_dispatch} \land \textit{Retry\_dispatch} \\
\hline
\end{array}
$$

The operations of the network manager are defined as composite operations involving operations from the nodes that send and receive mails. For brevity, only some of these operations are explained below:

The operation *AcceptMail* is invoked to send an acceptance of a mail to its sender. It is defined as a composition of two operations connected by the environment enrichment operator. The first operation in this composition is defined by an unnamed schema that declares the temporary variables used by the second operation. The latter itself is a composite operation and is described as follows: The sender sends a mail and waits for an acknowledgement. The network manager after receiving the mail from the sender (indicated by the right arrow in the parallel communication operator) checks whether the *regular queue* is full. If not full, the network manager adds the mail to the *regular queue* and puts the time stamp on the mail.

The operation *RefuseMail* is defined in a similar way except for the result of checking the *regular queue*. In this case, the network manager ensures that the *regular queue* is full and hence the mail is not accepted.

The purpose of the *Regular_displatch_successful* operation is to dispatch the mail at the front of the *regular queue*. It is defined as a composite operation. The first operation in the composition is an unnamed schema and it declares the temporary variables that are used by the second operation. The latter is a composite operation and is described as follows: The network manager first confirms that the *incoming tray* of the receiver of the mail is not full. If so, the network manager extracts the mail from the *regular queue* which is then received by the receiver.

In *Retry_dispatch_successful*, the network manager confirms that the *incoming tray* of the receiver is not full before extracting the mail. The network manager must also confirm that the time limit for the mail has not expired. This time constraint will be included in the filter (discussed in the next section) for this operation.

5.3 Filter Specifications - Second Tier of RTOZ

This section introduces the filter specifications for the classes in the mail system. The following are the global timing parameters:

$$t_{ack}, t_{dispatch}, t_{deliver} : Time$$

The variable t_{ack} denotes the time limit before which the network manager must send an acknowledgement to the sender, $t_{dispatch}$ is the time taken for the network manager to dispatch a mail successfully, and $t_{deliver}$ is the time limit before which a mail that was accepted by the network manager must be delivered to the receiver.

Filter for Mail Class There are three filters in the class *Mail_***filter** corresponding to the three operations in the class *Mail*. Each filter specifies the mapping of the functional variables to their respective timing variables. There are no additional timing constraints in any of these filters.

*Mail_***filter**

EditTo

$$to, to' : NODEID$$
$$t : RealTime$$
$$\delta t : Time$$
$$\mathbf{timemap} = \{to \mapsto t, to' \mapsto t + \delta t\}$$

EditBody

$$body, body' : NODEID$$
$$t : RealTime$$
$$\delta t : Time$$
$$\mathbf{timemap} = \{body \mapsto t, body' \mapsto t + \delta t\}$$

StampIt

$$timestamp, timestamp' : RealTime$$
$$ts? : RealTime$$
$$t : RealTime$$
$$\delta t, \epsilon_{1ts?} : Time$$
$$\mathbf{timemap} = \{timestamp \mapsto t, timestamp' \mapsto t + \delta t, ts? \mapsto t + \epsilon_{1ts?}\}$$

The filter specification *MailQueue_***filter** corresponds to the class *MailQueue*. There are no additional timing constraints introduced in *MailQueue_***filter**.

$MailQueue\_\mathbf{filter}$

$INIT$

$buffer : \mathrm{iseq}\ Mail$
$t : RealTime$

$\mathbf{timemap} = \{buffer \mapsto t\}$

$AddMail$

$buffer, buffer' : \mathrm{iseq}\ Mail$
$new? : Mail$
$t : RealTime$
$\delta t, \epsilon_{1new?} : Time$

$\mathbf{timemap} = \{buffer \mapsto t, buffer' \mapsto t + \delta t, new? \mapsto t + \epsilon_{1new?}\}$

$ExtractMail$

$buffer, buffer' : \mathrm{iseq}\ Mail$
$out! : Mail$
$t : RealTime$
$\delta t, \epsilon_{2out!} : Time$

$\mathbf{timemap} = \{buffer \mapsto t, buffer' \mapsto t + \delta t, out! \mapsto t + \epsilon_{2out!}\}$

$IsFull$

$buffer : \mathrm{iseq}\ Mail$
$result! : Mail$
$t : RealTime$
$\delta t, \epsilon_{2result!} : Time$

$\mathbf{timemap} = \{buffer \mapsto t, result! \mapsto t + \epsilon_{2result!}\}$

Filter for Node Class The filters *Send*, *ReceiveACK*, *ReceiveDeliveryMessage* and *UpdateMailbox* in the filter specification *Node_***filter** are straightforward and indicate only the time maps.

The filter *SendAndWait* indicates that the operation *ReceiveACK* in the composition must be invoked only after the *Send* operation (due to the sequential composition). In addition, the filter also specifies that the acknowledgement must be received within t_{ack} time units after the mail is submitted. Since the *Send* operation is invoked at time t, the acknowledgement must be received within $t + t_{ack}$ time units. The acknowledgement is recieved at time $\epsilon_{1ack?}$ from the invocation of the operation *ReceiveACK*; in this case, it is invoked at time $t + Send.\delta t$, and hence the constraint $Send.\delta t + ReceiveACK.\epsilon_{1ack?} \leq t_{ack}$ is introduced.

Other filters in *Node_***filter** are straightforward and can be interpreted according to the semantics of the filter specifications for composite operations. They do not introduce any additional timing constraints.

$$Node\_\mathbf{filter}$$

$$\begin{array}{l}
\underline{Send} \\
to? : NODEID \\
body? : MailContent \\
m! : Mail \\
t : RealTime \\
\delta t, \epsilon_{1to?}, \epsilon_{1body?}, \epsilon_{2m!} : Time \\
\hline
\mathbf{timemap} = \{to? \mapsto t + \epsilon_{1to?}, body? \mapsto \epsilon_{1body?}, m! \mapsto \epsilon_{2m!}\}
\end{array}$$

$$\begin{array}{l}
\underline{ReceiveACK} \\
ack? : Acknowledgement \\
t : RealTime \\
\delta t, \epsilon_{1ack?} : Time \\
\hline
\mathbf{timemap} = \{ack? \mapsto t + \epsilon_{1ack?}\}
\end{array}$$

$$\begin{array}{l}
\underline{ReceiveDeliveryMessage} \\
delivery? : DeliveryMessage \\
t : RealTime \\
\delta t, \epsilon_{1delivery?} : Time \\
\hline
\mathbf{timemap} = \{delivery? \mapsto t + \epsilon_{1delivery?}\}
\end{array}$$

$$\begin{array}{l}
SendAndWait \mathrel{\widehat=} \big[\, t : RealTime;\ \delta t : Time \,\big] \bullet \\
\quad (Send \fatsemi ReceiveACK) \bullet \big[\, Send.\delta t + ReceiveACK.\epsilon_{1ack?} \leq t_{ack} \,\big] \\
Receive \mathrel{\widehat=} \big[\, nid : NODEID;\ intray : MailQueue;\ new? : Mail; \\
\quad t : RealTime;\ \delta t, \epsilon_{1new?} : Time \,\big] \mid \\
\quad \mathbf{timemap} = \{nid \mapsto t, intray \mapsto t, new? \mapsto t + \epsilon_{1new?}\} \,\big] \bullet \\
\quad intray.AddMail
\end{array}$$

$$\begin{array}{l}
\underline{UpdateMailbox} \\
mbox, mbox' : NODEID \pfun \mathbb{P}\, Mail \\
m? : Mail \\
t : RealTime \\
\delta t, \epsilon_{1m?} : Time \\
\hline
\mathbf{timemap} = \{mbox \mapsto t, mbox' \mapsto t + \delta t, m? \mapsto t + \epsilon_{1m?}\}
\end{array}$$

$Node\_\mathbf{filter}(contd...)$

$$
\begin{array}{l}
Read \mathrel{\widehat=} [\, intray : MailQueue;\ out!, m? : Mail;\ t : RealTime;\ \delta t, \epsilon_{2out!}, \epsilon_{1m?} : Time \mid \\
\qquad \mathbf{timemap} = \{ intray \mapsto t, out! \mapsto t + \epsilon_{2out!}, m? \mapsto t + \epsilon_{1m?} \} \,] \bullet \\
\qquad intray.ExtractMail \bullet UpdateMailbox[out!/m?] \\
Forward \mathrel{\widehat=} [\, m?, m! : Mail;\ t : RealTime;\ \delta t, \epsilon_{1m?}, \epsilon_{2m!} : Time \mid \\
\qquad \mathbf{timemap} = \{ m? \mapsto t + \epsilon_{1m?}, m! \mapsto t + \epsilon_{2m!} \} \,] \bullet \\
\qquad Read \mathbin{;} m?.EditTo \mathbin{;} SendAndWait \\
Reply \mathrel{\widehat=} [\, m?, m! : Mail;\ t : RealTime;\ \delta t, \epsilon_{1m?}, \epsilon_{2m!} : Time \mid \\
\qquad \mathbf{timemap} = \{ m? \mapsto t + \epsilon_{1m?}, m! \mapsto t + \epsilon_{2m!} \} \,] \bullet \\
\qquad Read \mathbin{;} m?.EditBody \mathbin{;} SendAndWait
\end{array}
$$

Filter for NetworkManager Class The filters in *NetworkManager*_**filter** are all defined as composite filters. All the filters in this specification, except *Receive* and *Dispatch*, use the following structure: (i) An unnamed schema introduces the signature of all the functional variables and the timing variables in the filter, followed by the environment enrichment operator. This allows the variables to be used in the rest of the filter. (ii) The next filter indicates the time maps of the functional variables. (iii) A composite filter is constructed using the filters that correspond to the component operations. Each filter in the composition specifies its invocation time. This is based on the semantics of the composite operator used in the corresponding functional specification. (iv) The last part, if present, introduces additional timing constraints using an unnamed schema. Most of the filters in *NetworkManager*_**filter** are straightforward and no further explanation is given. Those filters that require special attention are discussed next.

$NetworkManager\_\mathbf{filter}$

INIT

$regular\_queue, retry\_queue : MailQueue$
$t : RealTime$

$\mathbf{timemap} = \{regular\_queue \mapsto t, retry\_queue \mapsto t\}$

$$\begin{array}{l}
AcceptMail \mathrel{\widehat=} \big[\, nodes : \mathbb{P}\, NODEID;\; regular_queue, retry_queue : MailQueue; \\
\quad ack! : Acknowledgement;\; result! : SpaceAvailability;\; m! : Mail;\; sender : Node; \\
\quad t : RealTime;\; \delta t, \epsilon_{2m!}, \epsilon_{2result!}, \epsilon_{2ack!} : Time \;\big| \\
\quad \mathbf{timemap} = \{nodes \mapsto t, regular_queue \mapsto t, retry_queue \mapsto t, \\
\qquad m! \mapsto t + \epsilon_{2m!}, ack! \mapsto t + \epsilon_{2ack!}, result! \mapsto t + \epsilon_{2result!}, sender \mapsto t\} \,\big] \bullet \\
\quad (sender.SendAndWait \overset{\parallel}{\rightleftarrows} \\
\qquad (regular_queue.IsFull \mathbin{⨟} (regular_queue.AddMail[m!/new?] \land m!.StampIt))) \bullet \\
\quad \big[\, \epsilon_{2ack!} > regular_queue.IsFull.\delta t \,\big] \\
RefuseMail \mathrel{\widehat=} \big[\, nodes : \mathbb{P}\, NODEID;\; regular_queue, retry_queue : MailQueue; \\
\quad ack! : Acknowledgement;\; result! : SpaceAvailability;\; sender : Node; \\
\quad t : RealTime;\; \delta t, \epsilon_{2result!}, \epsilon_{2ack!} : Time \;\big| \\
\quad \mathbf{timemap} = \{nodes \mapsto t, regular_queue \mapsto t, retry_queue \mapsto t, \\
\qquad ack! \mapsto t + \epsilon_{2ack!}, result! \mapsto t + \epsilon_{2result!}, sender \mapsto t\} \,\big] \bullet \\
\quad (sender.SendAndWait \overset{\parallel}{\rightleftarrows} regular_queue.IsFull) \bullet \\
\quad \big[\, \epsilon_{2ack!} > regular_queue.IsFull.\delta t \,\big] \\
Regular_dispatch_successful \mathrel{\widehat=} \\
\quad \big[\, nodes : \mathbb{P}\, NODEID;\; regular_queue : MailQueue; \\
\quad receiver : Node;\; tray : MailQueue;\; out! : Mail;\; result! : SpaceAvailability; \\
\quad t : RealTime;\; \delta t, \epsilon_{2result!}, \epsilon_{2out!} : Time \;\big| \\
\quad \mathbf{timemap} = \{nodes \mapsto t, regular_queue \mapsto t, receiver \mapsto t, tray \mapsto t, \\
\qquad out! \mapsto t + \epsilon_{2out!}, result! \mapsto t + \epsilon_{2result!}\} \,\big] \bullet \\
\quad (tray.IsFull \mathbin{⨟} (regular_queue.ExtractMail \overset{\parallel}{\rightarrow} receiver.Receive[out!/new?])) \bullet \\
\quad \big[\, \delta t \leq t_{dispatch} \land (t - out!.timestamp) \leq t_{deliver} \,\big]
\end{array}$$

$NetworkManager\_\mathbf{filter}(contd...)$

$$
\begin{array}{l}
Retry\_dispatch\_successful \mathrel{\widehat=} \\
\quad [\, nodes : \mathbb{P}\, NODEID;\ retry\_queue : MailQueue; \\
\quad receiver : Node;\ tray : MailQueue;\ out! : Mail;\ result! : SpaceAvailability; \\
\quad t : RealTime;\ \delta t, \epsilon_{2result!}, \epsilon_{2out!} : Time \mid \\
\quad \mathbf{timemap} = \{ nodes \mapsto t, retry\_queue \mapsto t, receiver \mapsto t, tray \mapsto t, \\
\qquad out! \mapsto t + \epsilon_{2out!}, result! \mapsto t + \epsilon_{2result!} \} \,] \bullet \\
\quad (tray.IsFull \mathbin{;} (retry\_queue.ExtractMail \overset{\parallel}{\rightarrow} receiver.Receive[out!/new?])) \bullet \\
\quad [\, \delta t \leq t_{dispatch} \land (t - out!.timestamp) \leq t_{deliver} \,] \\
Regular\_dispatch\_fail \mathrel{\widehat=} \\
\quad [\, nodes : \mathbb{P}\, NODEID;\ regular\_queue, retry\_queue : MailQueue; \\
\quad receiver : Node;\ tray : MailQueue;\ out! : Mail;\ result! : SpaceAvailability; \\
\quad t : RealTime;\ \delta t, \epsilon_{2result!}, \epsilon_{2out!} : Time \mid \\
\quad \mathbf{timemap} = \{ nodes \mapsto t, regular\_queue \mapsto t, retry\_queue \mapsto t, \\
\qquad receiver \mapsto t, tray \mapsto t, out! \mapsto t + \epsilon_{2out!}, result! \mapsto t + \epsilon_{2result!} \} \,] \bullet \\
\quad (tray.IsFull \mathbin{;} (regular\_queue.ExtractMail \overset{\parallel}{\rightarrow} retry\_queue.AddMail[out!/new?] \bullet \\
\quad [(t - out!.timestamp) \leq t_{deliver} \,] \\
Retry\_dispatch\_fail \mathrel{\widehat=} \\
\quad [\, nodes : \mathbb{P}\, NODEID;\ retry\_queue : MailQueue; \\
\quad receiver : Node;\ tray : MailQueue;\ out! : Mail;\ result! : SpaceAvailability; \\
\quad t : RealTime;\ \delta t, \epsilon_{2result!}, \epsilon_{2out!} : Time \mid \\
\quad \mathbf{timemap} = \{ nodes \mapsto t, retry\_queue \mapsto t, receiver \mapsto t, tray \mapsto t, \\
\qquad out! \mapsto t + \epsilon_{2out!}, result! \mapsto t + \epsilon_{2result!} \} \,] \bullet \\
\quad (tray.IsFull \mathbin{;} (retry\_queue.ExtractMail \overset{\parallel}{\rightarrow} retry\_queue.AddMail[out!/new?])) \bullet \\
\quad [(t - out!.timestamp) \leq t_{deliver} \,] \\
Delivered \mathrel{\widehat=} \\
\quad [\, nodes : \mathbb{P}\, NODEID;\ sender : Node;\ out! : Mail;\ delivery! : DeliveryMessage; \\
\quad t : RealTime;\ \delta t, \epsilon_{2out!}, \epsilon_{2delivery!} : Time \mid \\
\quad \mathbf{timemap} = \{ nodes \mapsto t, sender \mapsto t, out! \mapsto t + \epsilon_{2out!}, \\
\qquad delivery! \mapsto t + \epsilon_{2delivery!} \} \,] \bullet \\
\quad sender.ReceiveDeliveryMessage \bullet [(t - out!.timestamp \leq t_{deliver} \,]
\end{array}
$$

$NetworkManager\_\mathbf{filter}(contd...)$

$$
\begin{array}{l}
NotDelivered \mathrel{\widehat=} \\
\quad [\, nodes : \mathbb{P}\, NODEID;\ sender : Node;\ out! : Mail;\ delivery! : DeliveryMessage; \\
\quad t : RealTime;\ \delta t, \epsilon_{2out!}, \epsilon_{2delivery!} : Time \mid \\
\quad \mathbf{timemap} = \{ nodes \mapsto t, sender \mapsto t, out! \mapsto t + \epsilon_{2out!}, \\
\qquad delivery! \mapsto t + \epsilon_{2delivery!} \} \,] \bullet \\
\quad sender.ReceiveDeliveryMessage \bullet [\, (t - out!.timestamp) > t_{deliver} \,] \\
Regular\_dispatch \mathrel{\widehat=} \\
\quad [\, delivery! : DeliveryMessage;\ result! : SpaceAvailability; \\
\qquad t : RealTime;\ \delta t, \epsilon_{2delivery!}, \epsilon_{2result!} : Time \mid \\
\quad \mathbf{timemap} = \{ delivery! \mapsto t + \epsilon_{2deliver!}, result! \mapsto t + \epsilon_{2result!} \} \,] \bullet \\
\quad (Regular\_dispatch\_successful \fatsemi Delivered) \mathbin{[]} \\
\quad ([\, delivery! : DeliveryMessage \mid \mathbf{timemap} = \{ delivery! \mapsto t + \epsilon_{2delivery!} \} \,] \bullet \\
\qquad Regular\_dispatch\_fail) \\
Retry\_dispatch \mathrel{\widehat=} \\
\quad [\, delivery! : DeliveryMessage;\ result! : SpaceAvailability; \\
\qquad t : RealTime;\ \delta t, \epsilon_{2delivery!}, \epsilon_{2result!} : Time \mid \\
\quad \mathbf{timemap} = \{ delivery! \mapsto t + \epsilon_{2delivery!}, result! \mapsto t + \epsilon_{2result!} \} \,] \bullet \\
\quad (Retry\_dispatch\_successful \fatsemi Delivered) \mathbin{[]} \\
\quad ([\, delivery! : DeliveryMessage \mid \mathbf{timemap} = \{ delivery! \mapsto t + \epsilon_{2delivery!} \} \,] \bullet \\
\qquad Retry\_dispatch\_fail) \mathbin{[]} \\
\quad ([\, result! : SpaceAvailability \mid \mathbf{timemap} = \{ result! \mapsto t + \epsilon_{2result!} \} \,] \bullet NotDelivered) \\
Dispatch \mathrel{\widehat=} Regular\_dispatch \land Retry\_dispatch
\end{array}
$$

In the composite filter *AcceptMail*, the expression

$$regular\_queue.IsFull[t + \epsilon_{2m!} / regular\_queue.IsFull.t]$$

indicates that the start time of the operation *regular_queue.IsFull* (i.e., *regular_queue.IsFull.t*) is substituted by the expression $t + \epsilon_{2m!}$. Stated otherwise, the operation *regular_queue.IsFull* starts only after the mail $m!$ is output by the *sender.SendAndWait* operation. The latter starts at the same time when the composite operation *AcceptMail* is invoked. In a similar way, the operation *regular_queue.AddMail* is invoked only after the operation *regular_queue.IsFull* is completed. The constraint

$$[\, \epsilon_{2ack!} > regular\_queue.IsFull.\delta t \,]$$

indicates that the acknowledgement should be sent to the sender only after chekcing whether or not there is sufficient space in the *regular queue*. Other filters can be interpreted in a similar way.

The additional timing constraint in the filter *Regular_dispatch_successful* namely

$$[\, \delta t \leq t_{dispatch} \land (t - out!.timestamp) \leq t_{deliver} \,]$$

indicates that this composite operation must be performed within $t_{dispatch}$ time units (as warranted by the application). In addition, the operation will be successful only if the time limit for the delivery has not expired. Each mail accepted by the network manager carries the time at which the mail was accepted, and the expression $t - out!.timestamp$ indicates the duration in which the mail was waiting in the queue. This duration must be less than or equal to $t_{deliver}$ for successful dispatch. The same comments apply to the filter *Retry_dispatch_successful*.

The filters *Regular_dispatch_fail* and *Retry_dispatch_fail* can be interpreted similar to their complements *Regular_dispatch_successful* and *Retry_dispatch_successful*, but for the constraint $\delta t \leq t_{dispatch}$. This condition is not necessary in the failure operations because the mail is not dispatched.

The time out period for delivery is indicated in the filter *NotDelivered* as an additional timing constraint.

5.4 Illustration

This section illustrates the validity of the equation

$$\overline{Op} \hookrightarrow OP \equiv (\overline{Op_1} \hookrightarrow Op_1)\theta(\overline{Op_1} \hookrightarrow Op_2) \ldots\ldots (1)$$

where θ represents a composite operator and $Op \mathrel{\widehat=} Op_1 \theta Op_2$. To illustrate the semantics of composite filters, consider the composite operation $Read$ defined in the class $Node$. This operation is defined using the environment enrichment operator:

$$Read \mathrel{\widehat=} intray.ExtractMail \bullet UpdateMailbox[out!/m?]$$

For simplicity, the unnamed schema in the original definition of $Read$ is ignored for the current discussion.

First consider the application of composite filter $\overline{Read} \hookrightarrow Read$. Based on the timemap given in the filter $\overline{Read}$, one could infer that

$$\begin{array}{l} intray \mapsto t \Rightarrow intray.buffer \mapsto t \\ out! \mapsto t + \epsilon_{2out!} \\ mbox \mapsto ExtractMail.t + ExtractMail.\delta t \\ m? \mapsto ExtractMail.t + ExtractMail.\delta t + \epsilon_{1m?} \end{array}$$

where t denotes the start time of the operation $Read$ (which in this case is the start time of the operation $ExtractMail$), $\epsilon_{2out!}$ is the offset defined in $ExtractMail$ and $\epsilon_{1m?}$ is the offset defined in $UpdateMailbox$.

Now consider the application of individual filters to the respective operations and then the application of composition:

$$\overline{ExtractMail} \hookrightarrow intray.ExtractMail \bullet \overline{UpdateMailbox} \hookrightarrow UpdateMailbox$$

This results in the following time map:

$$intray \mapsto t_1 \Rightarrow intray.buffer \mapsto t_1$$
$$out! \mapsto t_1 + \epsilon_{2out!}$$
$$mbox \mapsto t_2$$
$$m? \mapsto t_2 + \epsilon_{1m?}$$

where t_1 denotes the start time of the operation $ExtractMail$, t_2 denotes the start time of the operation $UpdateMailbox$, $\epsilon_{2out!}$ is the offset defined in $ExtractMail$ and $\epsilon_{1m?}$ is the offset defined in $UpdateMailbox$. In the composition, the composite operation $Read$ starts at the same time $intray.ExtractMail$ starts and hence $t = t_1$, where t denotes the start time of $Read$. According to the semantics of the environment enrichment operator, the operation $UpdateMailbox$ starts after $intray.ExtractMail$ is completed and hence $t_2 = intray.ExtractMail.t + intray.ExtractMail.\delta t$.

It is not hard therefore to conclude that the semantics of composite filters satisfies the equation (1).

6 Conclusion and Future Work

The main contribution of the paper is the real-time specification language RTOZ, a conservative extension of Object-Z. The novelty of the language extension is in the addition of a new tier without altering the features of Object-Z. So the extension is not linear but orthogonal, yet the same Object-Z notation is used in both tiers.

In one tier of RTOZ untimed system is specified. The timing constraints governing the operations are specified in the second tier. There is a one-one correspondence between the specification units in the first tier and the timing constraints governing those units, called filters, in the second tier. Corresponding to each composition operator in the untimed tier a composition operator in the timed tier is defined. Semantics is given for composing filters, binding filters to their corresponding untimed operations, and applying filters to their corresponding operations. The interpretation for timed specification satisfies the general requirement that the composition of two timed systems is the timed system of the composition of the untimed components.

Some of the major advantages of RTOZ are summarized below:

- RTOZ provides state-based descriptions of the functionality in an application and their timing constraints in two separate tiers. Refinement of the first tier specifications do not affect the filters. Timing constraints can be modified or refined independent of changes to functional units.
- RTOZ separates the timing constraints from the functional specification thus allowing the user to concentrate on different aspects of the design independently. By virtue of object-oriented principles and separation of concerns, reuse potential is increased.
- RTOZ introduces very minimal additional syntactic notations; most of the notations in RTOZ are the same as Object-Z. Hence, learning and using RTOZ requires only knowledge of Object-Z.

- Timed specification is not explicitly given in RTOZ. Application of filters to their respective operations can be automated. This is a great advantage especially when dealing with composite operations.

- Based on the semantics, given in the logical style, a tool can be constructed to implement filter binding, filter application, and filter composition. Object-Z tools can be used in both tiers. Consequently, consistency of timed compositions can be mecahnically checked.

- RTOZ can be further extended to multi-tier specifications by introducing more tiers similar to the filter specification. For example, one can introduce a third tier that describes business constraints or resources constraints without altering the specifications in the other two tiers. Additional semantics is necessary to explain the mapping of the third tier variables with those on the other two tiers. This kind of orthogonalization will be quite helpful for developing the specification for large and complex systems.

RTOZ has been used to specify a robotic controller; see [2, 3]. A type checker [21] for RTOZ has been developed. Work is in progress to develop the other tools.

References

[1] M. Aksit *et al.*, "Real-Time Specification Inheritance Anomalies and Real-Time Filters", *ECOOP'94: European Conference on Object-Oriented Programming, Lectures Notes in Computer Science 821*, Springer-Verlag, 1994, pp. 386–405.

[2] V.S. Alagar and K. Periyasamy, " Real-Time Object-Z: A Language for the Specification and Design of Real-Time Reactive Systems", Technical Report, Department of Computer Science, Concordia University, Montreal, Canada, April 1996.

[3] V.S. Alagar and K. Periyasamy, "Off-Line Verification of an Intelligent Control", Chapter 9 in *Methods and Applications of Intelligent Control*, Ed. Spyros G. Tzafestas, Kluwer Academic Publishers, 1997.

[4] R. Alur and D.J. Dill, "A Theory of Timed Automata", *Theoretical Computer Science*, Vol. 126, 1994, pp. 183-235.

[5] R. Duke, G. Rose and G. Smith, "Object-Z: A Specification Language Advocated for the Description of Standards", *Computer Standards & Interfaces*, Vol. 17, 1995, pp. 511-533.

[6] C.J. Fidge and A. J. Wellings, "An Action-Based Formal Model for Concurrent Real-Time Systems", *Formal Aspects of Computing*, Vol. 3, 1996, pp. 1-34.

[7] C. Fidge, P. Kearney and M. Utting, "A Formal Method for Building Concurrent Real-Time Software", *IEEE Software*, 14(2), March/April 1997, pp. 99-106.

[8] C. Ghezzi, D. Mandioli and A. Morzenti, “TRIO: A Logic Language for Executable Specifications of Real-Time Systems”, *Journal of Systems and Software*, Vol. 12, No. 2, May 1990, pp. 107–123.

[9] D. Gray, *Introduction to the Formal Design of Real-Time Systems*, Applied Computing Series, Springer-Verlag, London, UK, 1999.

[10] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, London, 1985.

[11] B.P. Mahony and J.S. Dong, “Blending Object-Z and Timed CSP: An Introduction to TCOZ”, *Proceedings of the International Conference on Software Engineering (ICSE’98)*, Kyoto, Japan, April 1998, pp. 95-104.

[12] B.P. Mahony and I.J. Hayes, “A Case Study in Timed Refinement: A Mine Pump”, *IEEE Transactions on Software Engineering*, Vol. 18, No. 9, 1992, pp. 817-826.

[13] A.R. Milner, *Calculus of Communicating Systems. Lecture Notes in Computer Science 92*, Springer-Verlag, 1980.

[14] A. Morzenti and P. San Pierto, “ Object-Oriented Logical Specification of Time Critical Systems”, *ACM Transactions on Software Engineering and Methodology*, Vol. 3, No. 1, Jan 1994, pp. 56–98.

[15] S. Ren and G. Agha, “RTSynchronizer: Language Support for Real-Time Specifications in Distributed Systems”, *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, ACM Press, June 1995.

[16] S. Schneider and J, Davies, “A Brief History of Timed CSP”, *Theoretical Computer Science*, Vol. 39, 1995, pp. 138.

[17] A. Sherif and H. Jifeng, “Towards a Time Model for *Circus*”, *Proceedings of the International Conference on Formal Engineering Methods (ICFEM 2002)*, published as *Lecture Notes in Computer Science 2495*, Springer-Verlag, 2002, pp. 613-624.

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

[19] G. Smith, *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.

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

[21] T.S. Varadarajan, “Type Checker for RTOZ”, Masters Thesis (submitted), Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada, March 1997.

[22] N. Wirth, “Toward a Discipline of Real-Time Programming”, *Communications of the ACM*, Vol. 20, No. 8, Aug 1977, pp. 577-583.