Semantics for a Real-Time Object-Oriented Programming Language

Ichiro Satoh [*] Mario Tokoro [†]

satoh@mt.cs.keio.ac.jp mario@mt.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1, Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

This paper presents a framework to define a semantics for a real-time object-orientation programming language and to verify programs written in the language. The semantics is defined as the collection of translation rules that map the syntactic constructions of the language into expressions in a process calculus extended with the notion of time. By using the expressive capabilities of the calculus, the semantics can appropriately capture temporal features of the language such as timeout exception and execution time, and concurrent object-oriented features such as concurrency, class, and object creation. Particularly, since it can restrict the number of concurrent activities which take place simultaneously, it allows to analyze and predict temporal and behavioral properties of programs executed even with a number of processors smaller than the number of active objects. Through an example, we illustrate a method of analyzing real-time programs.

1 Introduction

Real-time systems are getting popular nowadays. Real-time systems have to control and interact many active devices such as other computers, sensors, and actuators. Besides, concurrent object-oriented computation [22, 24] is based on interactions with active entities. The concept of concurrent object-orientation is considered as a powerful methodology to design and implement real-time systems. Recently, many real-time programming languages have been proposed based on this concept, for example see [8, 19].

Real-time systems need to meet certain time constraints. The correctness of programs for real-time systems depends not only on the functionally behavioral results of computations but also on the time at which the results are produced. However, the dynamic features of concurrent object-orientation such as runtime object creation, concurrency, and message passing make it difficult to predict the temporal properties of systems. This is because these features seriously affect the execution time of programs and the number of concurrent activities. Also, the features highly depend on the behavioral properties of programs such as the contents of program variables. Consequently, to develop correct real-time systems based on the concept of concurrent objects, we need the support of an analysis for both the temporal and behavioral properties of the systems, particularly including the dynamical features of concurrent objects.

One typical approach is to develop higher abstraction models for reasoning about real-time systems based on petri nets, temporal logic, and process calculi. However, we believe that real-time systems should be specified and verified at the program level rather than at such higher levels, since the former is the notation where programs are written, debugged, and maintained.

In the past few years, several researchers have proposed frameworks to reason about programs described in (non object-oriented) real-time programming languages, based on denotational semantics and temporal logic. However, these frameworks cannot inherently capture most dynamical features of concurrent objects. Also, the execution models of these frameworks are rather unrealistic, for example the *maximum parallelism* which allows each activity to have its own processor, or the *interleaving* which allows at most one activity to be executed any time.

In this paper, we investigate a temporal semantics for a real-time programming language with dynamical features such as object (or process) creation and message passing. The objective of the semantics is to

[*]Partially supported by JSPS Fellowships for Japanese Junior Scientists

[†]Also with Sony Computer Science Laboratory Inc. 3-14-13 Higashi-Gotanda, Shinagawa-ku, Tokyo, 141, Japan

provide a general framework to predict and analyze both the temporal and the behavioral properties of programs written in such a langauge, through the direct analysis of program statements. Particularly, we take the number of concurrent activities into consideration, since the execution time of programs is highly dependent on the number of available processors.

We first propose a simple language $\mathcal{R}_{\mathcal{T}}$ which has peculiar features of real-time languages and concurrent object-oriented ones. We also adopt a process calculus as a meta-language for the semantics definition. This is because process calculus is a well-studied theory for concurrency and provides algebraic frameworks for analyzing and substituting objects. However, ordinary process calculi cannot model the temporal properties of real-time systems. We thus introduce a timed extended process calculus, called RtCCS [15], which is an extension of Milner's CCS [11] with the notion of quantitative time. Then, the semantics is defined in the form of translation rules for the syntactic constructions of the language. A translation rule for a construction encodes the behavioral and the temporal properties of the construction into corresponding expressions in RtCCS. We can strictly analyze $\mathcal{R}_{\mathcal{T}}$'s arbitrary programs through the theoretical framework of RtCCS.

The organization of this paper is as follows: In the next section we informally introduce a real-time object-oriented programming language, called $\mathcal{R}_{\mathcal{T}}$. Section 3 defines a timed extended process calculus, called RtCCS, as a meta-language for our semantics framework. In Section 4 we give the semantics of $\mathcal{R}_{\mathcal{T}}$ through the translation into the calculus. Section 5 surveys some related work. In Section 6, we present some concluding remarks. Finally in Appendix we present a sketch of an analysis of $\mathcal{R}_{\mathcal{T}}$ programs through an example.

2 $\mathcal{R}_{\mathcal{T}}$: a Real-Time Object-Oriented Language

This section summarizes the syntax and the informal semantics of $\mathcal{R}_{\mathcal{T}}$.

Syntax of $\mathcal{R}_{\mathcal{T}}$

The syntax of $\mathcal{R}_{\mathcal{T}}$ is defined in Figure 1. We use the following syntactic meta variables in the definition: *Prog* is a set of programs, *Class* a set of class declarations, *Stat* a set of statements, *Exp* a set of expressions, *Bool* a set of boolean expressions, *Dec* a set of variable declarations, C a class name, M a method name, and X a program variable name.

$$\begin{array}{lll}
\mathit{Prog} & ::= & \textbf{program}\ \mathit{Class}\ \textbf{is}\ \mathit{Dec}\ \textbf{in}\ \mathit{Stat}\ \textbf{endp} \\
\mathit{Class} & ::= & \textbf{class}\ C\ \textbf{is}\ \mathit{Dec}\ \textbf{new}\ \mathit{Stat}\ \mathit{Method}\ \textbf{endc} \\
 & | & \mathit{Class}\ \mathit{Class} \\
\mathit{Method} & ::= & \textbf{method}\ M(X)\ \textbf{is}\ \mathit{Dec}\ \textbf{in}\ \mathit{Stat}\ \textbf{endm} \\
 & | & \mathit{Method}\ \mathit{Method} \\
\mathit{Dec} & ::= & \textbf{var}\ X \\
 & | & \mathit{Dec}\ \mathit{Dec} \\
\mathit{Stat} & ::= & X := \mathit{Exp} \\
 & | & \textbf{return}\ \mathit{Exp} \\
 & | & \textbf{wait}\ \mathit{Exp} \\
 & | & \textbf{if}\ \mathit{Bool}\ \textbf{then}\ \mathit{Stat}\ \textbf{else}\ \mathit{Stat}\ \textbf{endif} \\
 & | & \textbf{while}\ \mathit{Bool}\ \textbf{do}\ \mathit{Stat}\ \textbf{od} \\
 & | & \mathit{Stat}\ ;\ \mathit{Stat} \\
\mathit{Exp} & ::= & 0|1|2|\cdots \\
 & | & X \\
 & | & \textbf{error} \\
 & | & C::\textbf{new} \\
 & | & X::M(\mathit{Exp})\ \textbf{timeout}\ \mathit{Exp} \\
\mathit{Bool} & ::= & \mathit{Exp}\ ==\ \mathit{Exp} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}$$

Figure 1: Syntax of $\mathcal{R}_{\mathcal{T}}$

Informal Semantics of $\mathcal{R}_{\mathcal{T}}$

We here give the informal semantics of $\mathcal{R}_{\mathcal{T}}$. An $\mathcal{R}_{\mathcal{T}}$ program consists of objects. An object is an instance which is created by its class[1]. Each object has an object identifier, variables to store its internal data, methods, and at most one activity (i.e. a single thread). The interaction among objects is based on the style of remote procedure call, but the called method can continue to be active after returning the result to the caller object. Moreover, in order to describe real-time properties, $\mathcal{R}_{\mathcal{T}}$ has a method call with timeout exception and a delay command to suspend programs for a specified time.

Program: A program construction *Prog* consists of class declarations, global program variable declarations, and a sequence of statements which is invoked at the beginning of program execution.

$$\textbf{program}\ \mathit{Class}\ \textbf{is}\ \mathit{Dec}\ \textbf{in}\ \mathit{Stat}\ \textbf{endp}$$

Class and Method Declaration: Classes serve as templates from which instance objects can be created.

[1]$\mathcal{R}_{\mathcal{T}}$ does not provide inheritance mechanism. This is because inheritance in concurrent object-oriented languages often causes unexpectable synchronization time, called *inheritance anomaly* (see, e.g. [10]). We need further investigation about inheritance mechanism for concurrent object-oriented languages with time constraints.

All objects in the same class contain the same structure: internal data and methods. A class declaration defines the class name, its instance variables, its initial statements, and its methods.

$$\begin{array}{l}\textbf{class}\ C\ \textbf{in}\ \mathit{Dec}\ \textbf{new}\ \mathit{Stat}_{new}\\ \quad\textbf{method}\ M_1(X_1)\ \textbf{is}\ \mathit{Dec}_1\ \textbf{in}\ \mathit{Stat}_1\ \textbf{endm}\\ \qquad\vdots\qquad\qquad\qquad\vdots\\ \quad\textbf{method}\ M_n(X_n)\ \textbf{is}\ \mathit{Dec}_n\ \textbf{in}\ \mathit{Stat}_n\ \textbf{endm}\\ \textbf{endc}\end{array}$$

where C is a class name, *Dec* the declarations of instance variables for each instance of the class, Stat_{new} a sequence of statements which is invoked at the creation of the instance, and M_i the name of a method of the instance. Dec_i is the local variable declarations of method M_i and Stat_i the body of method M_i.

Dynamic Object Creation: When the following expression is evaluated, an instance object is created according to the class definition of class C and then the expression returns the identifier of the created object.

$$C::\textbf{new}$$

Method Call with Timeout: In $\mathcal{R}_\mathcal{T}$'s method call, the caller is blocked until a reply is received from the called method. If the reception of the reply is within a specified time, the expression is evaluated as the reply value, otherwise as **error**.

$$X::M(\mathit{Exp}_1)\ \textbf{timeout}\ \mathit{Exp}_2$$

where a message M with an argument parameter Exp_1 is sent to an indicated object by the object identifier stored in X. The result of Exp_2 indicates the deadline time of the timeout.

Delaying Execution: In real-time systems, it is often useful to make programs wait for a specified period. $\mathcal{R}_\mathcal{T}$ provides an operation to suspend the execution as many time units as the value of the expression *Exp*.

$$\textbf{wait}\ \mathit{Exp}$$

Asynchronous Return: When the following statement is executed, the called method returns the value of expression *Exp* to its caller object and can continue to execute its following statements. Note that after the evaluation of this statement, the sender and receiver object can execute concurrently.

$$\textbf{return}\ \mathit{Exp}$$

Although $\mathcal{R}_\mathcal{T}$ has no asynchronous communication mechanism, this return can cause a communication to be viewed as an *asynchronous* one [2].

[2]Please note that the asynchronous return can realize the future synchronization [24].

3 RtCCS : A Meta Language for Temporal Semantics

This section introduces a meta-language called *Real-time Calculus of Communicating Systems* (RtCCS) for our semantics definition[3]. It is an extension of CCS [11] with with the ability of expressing the passage of time and behavior depending on time.

The passage of time is represented as a special action: $\surd$ called *tick* action. The action is a synchronous broadcast message over all processes. A tick action corresponds to the passage of one time unit. Also, to represent behaviors dependent on the advancing of time, we introduce a special binary operator: $\langle\ ,\ \rangle_t$, called a *timeout* operator. $\langle P, Q\rangle_t$ has the semantics of timeout handling. $\langle P, Q\rangle_t$ behaves as P if P can execute an initial transition within t time units, and behaves as Q if P does not perform any action within t time units.

Syntax of RtCCS

The syntax of RtCCS is essentially the same as that of CCS, except for two newly introduced temporal primitives: *tick action* $\surd$ and *timeout operator* $\langle\ ,\ \rangle_t$.

We presuppose that $\mathcal{A}$ is a set of communication action names and $\overline{\mathcal{A}}$ the set of co-names. Let a,b,... range over $\mathcal{A}$ and $\overline{a}$,$\overline{b}$,... over $\overline{\mathcal{A}}$. An action $\overline{a}$ is the complementary action of a and $\overline{\overline{a}} \equiv a$. Let $\mathcal{L} \equiv \mathcal{A} \cup \overline{\mathcal{A}}$ be a set of action labels ranged over $\ell, \ell', \ldots$. Let τ denote an internal action, and $\surd$ a tick action. Finally let $\mathit{Act} \equiv \mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$, ranged over by α,β,....

The set $\mathcal{E}$ of RtCCS expressions ranged over by $E, E_1, E_2, \ldots$ is defined recursively by the following abstract syntax:

$$\begin{array}{rcl} E & ::= & \mathbf{0} \ \mid\ X \ \mid\ \alpha.E \ \mid\ E_1 + E_2 \ \mid\ E_1|E_2 \\ & \mid & E[f] \ \mid\ E \setminus L \ \mid\ \mathbf{rec}\, X : E \ \mid\ \langle E_1, E_2\rangle_t \end{array}$$

where t is an element of the set of natural number containing an element 0 and f is a relabeling function, $f \in \mathit{Act} \rightarrow \mathit{Act}$ and $f(\tau) = \tau$. We assume that L is a subset of $\mathcal{L}$, and that X is always *guarded* [4]. For convention, instead of $\mathbf{rec}\, X : E$, we shall often use the more readable notation $X \stackrel{\rm def}{=} E$. We denote the set of closed expressions by $\mathcal{P}(\subseteq \mathcal{E})$, ranged over P, Q.

Intuitively, the meanings of constructions are as follows: $\mathbf{0}$ represents a terminated process; $\alpha.E$ performs

[3]In this paper the calculus is given only in outline. For details please consult the authors' other paper [15].

[4]X is *guarded* in E if each occurrence of X is only within some subexpressions $\alpha.E'$ in E where α is not an empty element; c.f. *unguarded* expressions, e.g. $\mathbf{rec}XX$ or $\mathbf{rec}XX + E$.

an action α and then behaves like E; $E_1 + E_2$ is the process which may behave as E_1 or E_2; $E_1|E_2$ represents processes E_1 and E_2 executing concurrently; $E[f]$ behaves like E but with the actions relabeled by function f; $E \setminus L$ behaves like E but with actions in $L \cup \bar{L}$ prohibited; $\mathbf{rec}\, X : E$ binds the free occurrences of X in E.

Operational Semantics of RtCCS

The semantics of RtCCS is given as a labeled transition system $\langle\, \mathcal{E},\ Act \cup \{\surd\},\ \{ \xrightarrow{\mu} \mid \mu \in Act \cup \{\surd\} \}\, \rangle$ where $\xrightarrow{\mu}$ is a transition relation ($\xrightarrow{\mu} \subseteq \mathcal{E} \times \mathcal{E}$). It is structurally given in two phases. The first phase defines the relation $\xrightarrow{\alpha}$ for each $\alpha \in Act$. The inference rules determining $\xrightarrow{\alpha}$ are given in Figure 2. The second phase defines the relation $\xrightarrow{\surd}$ by inference rules given in Figure 3.

$$\frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 + E_2 \xrightarrow{\alpha} E_1'}$$

$$\frac{E_2 \xrightarrow{\alpha} E_2'}{E_1 + E_2 \xrightarrow{\alpha} E_2'} \qquad \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1|E_2 \xrightarrow{\alpha} E_1'|E_2}$$

$$\frac{E_2 \xrightarrow{\alpha} E_2'}{E_1|E_2 \xrightarrow{\alpha} E_1|E_2'} \qquad \frac{E_1 \xrightarrow{a} E_1',\ E_2 \xrightarrow{\overline{a}} E_2'}{E_1|E_2 \xrightarrow{\tau} E_1'|E_2'}$$

$$\frac{E \xrightarrow{\alpha} E',\ \alpha \notin L \cup \overline{L}}{E \setminus L \xrightarrow{\alpha} E' \setminus L} \qquad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\frac{E\{\mathbf{rec}\, X : E/X\} \xrightarrow{\alpha} E'}{\mathbf{rec}\, X : E \xrightarrow{\alpha} E'} \qquad \frac{E_1 \xrightarrow{\alpha} E_1'\ ,\ t > 0}{\langle E_1, E_2 \rangle_t \xrightarrow{\alpha} E_1'}$$

Figure 2: Transition Rules for $\xrightarrow{\alpha}$

$$\frac{}{\ell.E \xrightarrow{\surd} \ell.E} \qquad \frac{}{\mathbf{0} \xrightarrow{\surd} \mathbf{0}}$$

$$\frac{E_1 \xrightarrow{\surd} E_1',\ E_2 \xrightarrow{\surd} E_2'}{E_1 + E_2 \xrightarrow{\surd} E_1' + E_2'} \qquad \frac{E_1 \xrightarrow{\surd} E_1',\ E_2 \xrightarrow{\surd} E_2',\ E_1|E_2 \not\xrightarrow{\tau}}{E_1|E_2 \xrightarrow{\surd} E_1'|E_2'}$$

$$\frac{E \xrightarrow{\surd} E'}{E \setminus L \xrightarrow{\surd} E' \setminus L} \qquad \frac{E \xrightarrow{\surd} E'}{E[f] \xrightarrow{\surd} E'[f]}$$

$$\frac{E\{\mathbf{rec}\, X : E/X\} \xrightarrow{\surd} E'}{\mathbf{rec}\, X : P \xrightarrow{\surd} P'} \qquad \frac{E_1 \xrightarrow{\surd} E_1'\ ,\ t > 0}{\langle E_1, E_2 \rangle_t \xrightarrow{\surd} \langle E_1', E_2 \rangle_{t-1}}$$

$$\frac{E_2 \xrightarrow{\mu} E_2'}{\langle E_1, E_2 \rangle_0 \xrightarrow{\mu} E_2'}$$

Figure 3: Transition Rules for $\xrightarrow{\surd}$

We will hereafter use the passage of data values between RtCCS expressions. RtCCS with value passing is easily derived from ordinary RtCCS by the following syntactically translation. We assume that D is the set of data values and that there are value variables x, y and a value expression d. For each expression E without free value variables, it is translated to $\widehat{E}$ as shown in Figure 4.

E	$\widehat{E}$
$a(x).E$	$\sum_{d\in D} a_d.\ \widehat{E[d/x]}$
$\overline{a}(d).E$	$\overline{a_d}.\widehat{E}$
$\tau.E$	$\tau.\widehat{E}$
$\sum_{i\in I} E_i$	$\sum_{i\in I} \widehat{E_i}$
$E_1\|E_2$	$\widehat{E_1}\|\widehat{E_2}$
$E[f]$	$\widehat{E}[\widehat{f}]$ where $\widehat{f}(l_d) = f(l)_d$
$E \setminus L$	$\widehat{E} \setminus \{l_d \| l \in L, d \in D\}$
$\langle E_1, E_2 \rangle_t$	$\langle \widehat{E_1}, \widehat{E_2} \rangle_t$
if *true* then E_1 else E_2	$\widehat{E_1}$
if *false* then E_1 else E_2	$\widehat{E_2}$

Figure 4: Translation of Expressions with Value into RtCCS Expressions

4 Translation Semantics for $\mathcal{R_T}$ Based on RtCCS

We here give the formal semantics of $\mathcal{R_T}$ as the collection of the translation rules mapping the syntactic constructions of $\mathcal{R_T}$ into expressions in RtCCS. The main source of inspiration is the semantics of a parallel imperative language $\mathcal{M}$ given by R. Milner in Chapter 8 of his book [11]. However, our work differs in the following ways: our target language is a real-time object-oriented language and our semantics can capture the temporal properties of the language as well as its behavioral ones.

Particularly, in order to ensure that real-time programs satisfy their time constraints, we need to predict the execution time of programs. Therefore, in our semantics the expressiveness of execution time is introduced. In this paper we assume that execution time for every statement in *Dec* and *Stat* is t_{stat} units of time[5]. Also, the execution time of a concurrent program is highly dependent on the number of physical processors. We introduce a restriction on the number of activities which may take place simultaneously.

Combinators

When translating the $\mathcal{R_T}$ constructions into RtCCS, we will use two combinators: "*Into*(x)" and "*Before*" as defined below. Particularly, the latter plays a key role of our temporal semantics.

The result of evaluating *Bool* and *Exp* expression is always communicated via res. The combinator

[5]Note taht our semantics can be easily adapted to the more general case.

"$Into(x)$" is used to transmit its value to the right process.

$$P\ Into(x)\ Q \overset{\text{def}}{=} (P|\mathsf{res}(x).Q) \setminus \{\mathsf{res}\}$$

$$\begin{aligned} P\ Before\ Q &\overset{\text{def}}{=} \mathsf{run}.(P[\mathsf{b}/\mathsf{done}]| \\ &\qquad \mathsf{b}.\langle \mathbf{0}, \overline{\mathsf{idle}}.Q\rangle_{t_{stat}}) \setminus \{\mathsf{b}\} \\ Processor &\overset{\text{def}}{=} \overline{\mathsf{run}}.\mathsf{idle}.Processor \\ Machine_N &\overset{\text{def}}{=} \underbrace{Processor|\cdots|Processor}_{N} \end{aligned}$$

Above t_{stat} is the execution time of a construction in *Stat* and *Dec*. We will often omit parentheses in such a way that, for example, "$P\ Before\ Q\ Before\ R$" denotes "$P\ Before\ (Q\ Before\ R)$"[6]. Precedence of the combinators is "$Into(x)$", "*Before*" in that order.

The combinator "*Before*" is used for modeling the following three properties: sequential concatenation, the execution time of a statement, and the restriction on the number of concurrent activities.

(1) We adopt output action $\overline{\mathsf{done}}$ as a signal of successful termination of every expression representing *Stat* and *Dec* statement. The combinator must receive input action done from its left expression before it invokes its right expression. Consequently, the left expression terminates and then the right one can be executed.

(2) By using RtCCS's timeout operator $\langle\ ,\ \rangle_t$, the combinator suspends its right expression at least for t_{stat} time units after the reception of done from its left one. This suspension represents the execution cost of the statement on the left.

(3) The combinator must receive action run from a *Processor* before it invokes its left process. Also, a *Processor* which has already sent $\overline{\mathsf{run}}$ to a *Before* combinator, cannot send $\overline{\mathsf{run}}$ to another combinator before receiving idle. As a result, the number of *Processor* is as well as the maximal number of *Before* whose left expressions can be executed. $Machine_N$ corresponds to a parallel computer with N processors.

We demonstrate a derivation of expressions representing three statements executed concurrently by two processors as follows:

$$\begin{aligned} & Machine_2|(P_1\ Before\ Q_1)| \\ &\quad (P_2\ Before\ Q_2)|(P_3\ Before\ Q_3) \\ \xrightarrow{\tau\,(\mathsf{run})}\ & \mathsf{idle}.Processor|\overline{\mathsf{run}}.\mathsf{idle}.Processor| \\ &\quad ((P_1[\mathsf{b}/\mathsf{done}]|\mathsf{b}.\langle \mathbf{0}, \overline{\mathsf{idle}}.Q_1\rangle_{t_{stat}}) \setminus \{\mathsf{b}\})| \\ &\qquad (P_2\ Before\ Q_2)|(P_3\ Before\ Q_3) \\ \xrightarrow{\tau\,(\mathsf{run})}\ & \mathsf{idle}.Processor|\mathsf{idle}.Processor| \\ &\quad ((P_1[\mathsf{b}/\mathsf{done}]|\mathsf{b}.\langle \mathbf{0}, \overline{\mathsf{idle}}.Q_1\rangle_{t_{stat}}) \setminus \{\mathsf{b}\})| \\ &\qquad ((P_2[\mathsf{b}/\mathsf{done}]|\mathsf{b}.\langle \mathbf{0}, \overline{\mathsf{idle}}.Q_2\rangle_{t_{stat}}) \setminus \{\mathsf{b}\})| \\ &\qquad\quad (P_3\ Before\ Q_3) \\ \longrightarrow\ & \cdots \end{aligned}$$

where P_1, P_2, P_3 are expressions representing statements. They indicate their termination at $\overline{\mathsf{done}}$. Only two of the expressions can become active and the other must suspend at least until one of the active expressions terminates.

Remarks

- Note that *Before* does not have to receive run from the same *Processor*. Consequently, statements in the same instances may be executed by different *Processors*.
- Since in RtCCS an executable communication (including τ) always precedes $\surd$ action, a communication must be executed as soon as it becomes executable. Consequently, if there is a *Processor* which is ready to send $\overline{\mathsf{run}}$, a statement waiting run can be executed immediately without unnecessary idling.
- Indeed our *Before* seems to be the same as the *Before* combinators of similar works [11, 14, 20, 21], in which it is used as a sequential composition between two expressions representing statements. However, our combinator is unique in representing the execution time of a statement and processor assignment.
- The reader may imagine a scenario where more than N processes execute concurrently because in *Before* combinators nested applications, the final process is never locked[7]. However, in our semantics definition such a final process always represents a semantic element with trivial computational cost. This problem can actually be ignored.

Translation Rules

We now give the semantics by the translation rule $[\![\,]\!]$ defined for each individual syntactic construction in $\mathcal{R}_{\mathcal{T}}$. We first define some notations used in our semantics. Let $Id \overset{\text{def}}{=} \{0, 1, \ldots\}$ be the set of the identifiers of instance objects. Let $Int \overset{\text{def}}{=} \{\hat{0}, \hat{1}, \ldots, \hat{n}, \ldots\}$ be the set of integers.

[6]Note that $P\ Before\ \mathbf{0} \neq P$ and $P\ Before\ (Q\ Before\ R) \neq (P\ Before\ Q)\ Before\ R$, unlike the *Before* combinators developed in [11, 14, 20, 21].

[7]Suppose $(\cdots\ Before\ R_1)|(\cdots\ Before\ R_2)|(\cdots\ Before\ R_3)$. R_1, R_2, R_3 can be executed concurrently even on two processors.

$$
\begin{array}{rcll}
[\![n]\!] & = & \overline{\mathsf{res}}(\,\hat{n}\,).\mathbf{0} \qquad n \in \mathit{Int} & \text{(E1)}\\
[\![X]\!] & = & \mathsf{read}_X(x).\overline{\mathsf{res}}(x).\mathbf{0} & \text{(E2)}\\
[\![\mathbf{error}]\!] & = & \overline{\mathsf{res}}(\mathit{error}).\mathbf{0} & \text{(E3)}\\
[\![C\!::\mathbf{new}]\!] & = & \overline{\mathsf{new}_C}.\mathsf{id}_C(i).\overline{\mathsf{res}}(i).\mathbf{0} & \text{(E4)}\\
[\![X\!::\!M(\mathit{Exp}_1)\ \mathbf{timeout}\ \mathit{Exp}_2]\!] & = & [\![\mathit{Exp}_2]\!]\ \mathit{Into}(t)([\![\mathit{Exp}_1]\!]\ \mathit{Into}(x)(\mathsf{read}_X(i). & \\
 & & (\langle\overline{\mathsf{call}_{i,M}}(x).\mathsf{ret}_{i,M}(y).\overline{\mathsf{reply}}(y).\mathbf{0}\ ,\ \overline{\mathsf{abort}}.\mathbf{0}\rangle_t\ | & \\
 & & \langle\mathsf{reply}(z).\overline{\mathsf{res}}(z).\mathbf{0}\ ,\ \overline{\mathsf{res}}(\mathit{error}).(\mathsf{abort}.\mathbf{0} + \mathsf{reply}(z).\mathbf{0})\rangle_t) & \\
 & & \backslash\{\mathsf{reply},\mathsf{abort}\})) & \text{(E5)}\\
 & & & \\
[\![\mathbf{true}]\!] & = & \overline{\mathsf{res}}(\mathit{true}).\mathbf{0} & \text{(B1)}\\
[\![\mathbf{false}]\!] & = & \overline{\mathsf{res}}(\mathit{false}).\mathbf{0} & \text{(B2)}\\
[\![\mathit{Exp}_1 == \mathit{Exp}_2]\!] & = & [\![\mathit{Exp}_1]\!]\ \mathit{Into}(x_1)\ ([\![\mathit{Exp}_2]\!]\ \mathit{Into}(x_2)(\overline{\mathsf{res}}(\mathit{equal}(x_1,x_2)).\mathbf{0})) & \text{(B3)}\\
 & & \text{where} \qquad \mathit{equal}(x_1,x_2) = \begin{cases} \mathit{true} & \text{if } x_1 = x_2\\ \mathit{false} & \text{otherwise}\end{cases} &
\end{array}
$$

Figure 5: Translation of Expressions in *Exp* and *Bool*

$$
\begin{array}{rcll}
[\![X := \mathit{Exp}]\!] & = & [\![\mathit{Exp}]\!]\ \mathit{Into}(x)\ \overline{\mathsf{write}_X}(x).\overline{\mathsf{done}}.\mathbf{0} & \text{(S1)}\\
[\![\mathbf{return}\ \mathit{Exp}]\!] & = & [\![\mathit{Exp}]\!]\ \mathit{Into}(x)\ \overline{\mathsf{ans}}(x).\overline{\mathsf{done}}.\mathbf{0} & \text{(S2)}\\
[\![\mathbf{wait}\ \mathit{Exp}]\!] & = & [\![\mathit{Exp}]\!]\ \mathit{Into}(t)\ \langle\mathbf{0},\overline{\mathsf{done}}.\mathbf{0}\rangle_t & \text{(S3)}\\
[\![\mathbf{if}\ \mathit{Bool}\ \mathbf{then}\ \mathit{Stat}_1\ \mathbf{else}\ \mathit{Stat}_2\ \mathbf{endif}]\!] & = & [\![\mathit{Bool}]\!]\ \mathit{Into}(x)\ \mathsf{if}\ x\ \mathsf{then}\ [\![\mathit{Stat}_1]\!]\ \mathsf{else}\ [\![\mathit{Stat}_2]\!] & \text{(S4)}\\
[\![\mathbf{while}\ \mathit{Bool}\ \mathbf{do}\ \mathit{Stat}\ \mathbf{od}]\!] & = & W & \text{(S5)}\\
\text{where} \quad W & \overset{\text{def}}{=} & [\![\mathit{Bool}]\!]\ \mathit{Into}(x)\ \mathsf{if}\ x\ \mathsf{then}\ [\![\mathit{Stat}]\!]\ \mathit{Before}\ W\ \mathsf{else}\ \overline{\mathsf{done}}.\mathbf{0} & \\
[\![\mathit{Stat}_1\ ;\ \mathit{Stat}_2]\!] & = & [\![\mathit{Stat}_1]\!]\ \mathit{Before}\ [\![\mathit{Stat}_2]\!] & \text{(S6)}
\end{array}
$$

Figure 6: Translation of Statements in *Stat*

Variable Declaration: We give the translation rules of variable declarations as below. The meaning of the declaration of a variable X is to create a storage location for X. The location is represented as a RtCCS process Loc_X. A value or an object identifier can be stored in the variable by sending a new value y on action write_X. Reading of a value from X is accomplished by receiving the stored value x from action $\overline{\mathsf{read}_X}$. We here define the access sort of variable. Let $L_X \overset{\text{def}}{=} \{\mathsf{read}_X, \mathsf{write}_X\}$ be the access sort of Loc_X and $L_{\mathit{Dec}} \overset{\text{def}}{=} \bigcup L_X$ where X is declared in *Dec*.

$$
\begin{array}{rcl}
[\![\mathbf{var}\ X]\!] & = & \mathit{Loc}_X | \overline{\mathsf{done}}.\mathbf{0}\\
\mathit{Loc}_X & \overset{\text{def}}{=} & \mathsf{write}_X(x).\mathit{Loc'}_X(x)\\
\mathit{Loc'}_X(x) & \overset{\text{def}}{=} & \mathsf{write}_X(y).\mathit{Loc'}_X(y)\\
 & & +\,\overline{\mathsf{read}_X}(x).\mathit{Loc'}_X(x)\\
 & & \\
[\![\mathit{Dec}_1\ \mathit{Dec}_2]\!] & = & [\![\mathit{Dec}_1]\!]\ \mathit{Before}\ [\![\mathit{Dec}_2]\!]
\end{array}
$$

To give a feel for our translation semantics we show that "**var** X **var** Y $\cdots$" is encodes into RtCCS expressions by using the translation rules as follows:

$$
\begin{array}{rl}
 & [\![\mathbf{var}\,X\ \ \mathbf{var}\,Y\ \cdots]\!]\\
= & [\![\mathbf{var}\,X]\!]\ \mathit{Before}\ [\![\mathbf{var}\,Y]\!]\ \mathit{Before}\ [\![\cdots]\!]\\
= & \mathsf{run}.(\mathit{Loc}_X|\overline{\mathsf{done}}.\mathbf{0}[\mathsf{b}/\mathsf{done}]|\mathsf{b}.\langle\mathbf{0}\ ,\ \overline{\mathsf{idle}}.\\
 & \quad \mathsf{run}.(\mathit{Loc}_Y|\overline{\mathsf{done}}.\mathbf{0}[\mathsf{b}/\mathsf{done}]|\mathsf{b}.\langle\mathbf{0}\ ,\ \overline{\mathsf{idle}}.[\![\cdots]\!]\rangle_{t_{stat}})\\
 & \qquad \backslash\{\mathsf{b}\}\rangle_{t_{stat}}) \setminus \{\mathsf{b}\}
\end{array}
$$

From the translated expression we can analyze the temporal properties of the variable declarations. For example, we can know that the execution of "**var** X **var** Y" takes two t_{stat} time units, since the expression contains two $\mathsf{run}.\langle\mathbf{0},\overline{\mathsf{idle}}.\cdots\rangle_{t_{stat}}$ sequences.

Expression: The translation rules of *Exp* and *Bool* constructions are listed in Figure 5. Every process representing a construction *Exp* and *Bool* terminates by yielding up its result through the output action $\overline{\mathsf{res}}$ into an *Into* combinator.

(E4) requests a process representing the class declaration of class C to create an instance of C via $\overline{\mathsf{new}_C}$, and then it receives the identifier of the created instance via id_C.

Next we consider the translation of $\mathcal{R}_\mathcal{T}$'s method call with timeout exception. There are two timeout exception cases: the method cannot be invoked within the deadline time, and the method has been invoked but cannot return its result within the deadline time. In the latter case, the caller must receive the result even after it timeouts. These exceptions are represented by using two RtCCS's $\langle\ ,\ \rangle_t$.

$$
\begin{aligned}
[\![\textbf{class } C \textbf{ is } \mathit{Dec} \textbf{ new } \mathit{Stat}\ \mathit{Method} \textbf{ endc}]\!] &= \mathit{ClassDec}_C \qquad \text{where} && \text{(C1)}\\
\mathit{ClassDec}_C &\overset{\text{def}}{=} \mathsf{new}_C.\mathsf{getid}(i).\overline{\mathsf{id}_C}(i).\\
&\quad\ (\mathit{ClassDec}_C|([\![\mathit{Dec}]\!]\ \mathit{Before}\ [\![\mathit{Stat}]\!]\ \mathit{Before}\ \mathit{Body}_{C,i}) \setminus L_{Dec})\\
\mathit{Body}_{C,i} &\overset{\text{def}}{=} ([\![\mathit{Method}]\!](C,i)|\mathsf{endm}.\mathit{Body}_{C,i}) \setminus \{\mathsf{endm}\}\\
[\![\mathit{Class}_1\ \mathit{Class}_2]\!] &= [\![\mathit{Class}_1]\!]|[\![\mathit{Class}_2]\!] && \text{(C2)}
\end{aligned}
$$

Figure 7: Translation Rules of Class Declarations in *Class*

$$
\begin{aligned}
&[\![\textbf{method } M(X) \textbf{ is } \mathit{Dec} \textbf{ in } \mathit{Stat} \textbf{ endm}]\!](C,i)\\
&\qquad = \mathsf{call}_{i,M}(x).(\mathsf{ans}(y).\overline{\mathsf{ret}_{i,M}}(y).\mathbf{0}|\mathit{Loc}_X|\overline{\mathsf{write}_X}(x). && \text{(M1)}\\
&\qquad\quad\ ([\![\mathit{Dec}]\!]\ \mathit{Before}\ [\![\mathit{Stat}]\!]\ \mathit{Before}\ \overline{\mathsf{endm}}.\mathbf{0}) \setminus L_{Dec}) \setminus L_X \setminus \{\mathsf{ans}\} && \text{(M1)}\\
&[\![\mathit{Method}_1 \cdots \mathit{Method}_n]\!](C,i) = [\![\mathit{Method}_1]\!](C,i) + \cdots + [\![\mathit{Method}_n]\!](C,i) && \text{(M2)}
\end{aligned}
$$

Figure 8: Translation Rules of Method Declarations in *Method*

The translated expression first evaluates Exp_2 and Exp_1 and then performs read_X to obtain the identifier i of the destination object from Loc_X. Then, it invokes method M of an instance with the identifier by sending $\overline{\mathsf{call}_{i,M}}$. If the invocation is possible within the deadline time, the first timeout operator waits the result at $\mathsf{ret}_{i,M}$ and then transmits the result to the second one via $\overline{\mathsf{reply}}$, otherwise it sends $\overline{\mathsf{abort}}$. If the second recieves the result at reply within the deadline time, it yields the result at $\overline{\mathsf{res}}$, otherwise it yields *error*.

Statement: The translation rules for *Stat* constructions are given in Figure 6. In (S1), the value of the evaluation of *Exp* is stored in Loc_X via $\overline{\mathsf{write}_X}$. In (S2), the value of the evaluation of *Exp* is sent to a process representing the called method. In (S3), $\langle \mathbf{0}, \overline{\mathsf{done}}.\mathbf{0}\rangle_t$ suspends $\overline{\mathsf{done}}$ to be performed for t units of time.

Class Declaration: The translation of class declaration is given in Figure 7. Class C is represented by a process which receives a request for instance creation at new_C and obtains a new identifier through getid from process *ObjId*. Then, it returns the identifier of the new instance through $\overline{\mathsf{id}_C}$ and creates $([\![\mathit{Dec}]\!]\ \mathit{Before}\ [\![\mathit{Stat}]\!]\ \mathit{Before}\ \mathit{Body}_{C,i}) \setminus L_{Dec}$ which corresponds to an instance of class C. Note that the instance variables are encapsulated by restriction "$\setminus L_{Dec}$".

Method Declaration: Figure 8 defines the translation of method declarations. The methods of an instance are invoked by receiving $\mathsf{call}_{i,M}$. The method body process first stores the argument value x of $\mathsf{call}_{i,M}$ in its local variable Loc_X and then executes $[\![\mathit{Stat}]\!]$. After receiving the return value at ans from (S2), it transmits the value to the caller through $\overline{\mathsf{ret}_{i,M}}$ and then continues into the following statement. Note that the translation of a sequence $Method_1 \cdots Method_n$ allows at most one execution in an instance to take place concurrently.

Program: Finally, we define the translation of program declaration as shown in Figure 9. $ObjId$ is a name server which generates a unique identifier for each object.

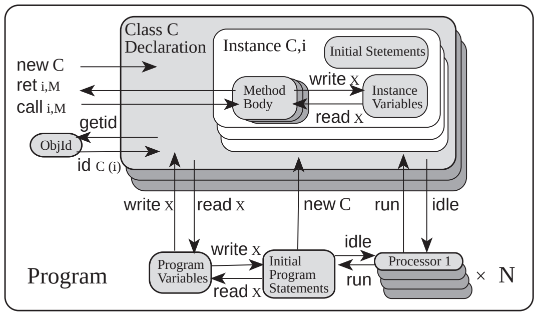

Figure 10: The Structure of Translation of *Prog*

Remarks

- By the ability of restricting the number of concurrent activities, it can easily capture various execution models. For example, when $N = 1$, it can simulate the *interleaving* and when $N = \infty$, the *maximum parallelism.*
- When there are less processors than the number of active objects, a scheduling policy is required to make good use of the available processors and resources. The timing properties of real-time systems highly depend on such a scheduling. For simplicity, our semantics however makes minimal assumptions about the scheduling and lacks the concept of fairness. The concept of scheduling and priority is left to our future study.

$$\begin{aligned}
&[\![\textbf{program } \mathit{Class} \textbf{ is } \mathit{Dec} \textbf{ in } \mathit{Stat} \textbf{ endp}]\!] \\
&\qquad = [\![\mathit{Class}]\!] | \mathit{ObjId}_0 | \mathit{Machine}_N | ([\![\mathit{Dec}]\!] \mathit{ Before } [\![\mathit{Stat}]\!] \mathit{ Before } \mathbf{0}) \setminus L_{\mathit{Dec}} \qquad \text{(P1)} \\
&\text{where} \quad \mathit{ObjId}_i \overset{\text{def}}{=} \overline{\mathsf{getid}}(i).\mathit{ObjId}_{i+1}
\end{aligned}$$

Figure 9: Translation of Program in *Prog*

5 Related Work

This section presents an overview of related work. Several denotational semantics have been explored for reasoning about real-time languages based on CSP [5][8], for example in [6, 7, 9, 18]. In these semantics, the behavior and timing of programs are associated with elements of linear history records. The execution models of the semantics developed in [9, 7] are unfortunately based on unrealistic models: the *maximum parallelism* or the *interleaving* and thus cannot represent the execution of processes sharing processors. Also, in [18] a *pseudo* upper bound of simultaneous activities is realized by introducing the utility rate of processors. On the other hand, our semantics can arbitrarily restrict the number of activities that can take place simultaneously. In [6] the author investigated denotational semantics for real-time processes sharing processors. However, the semantics assumes that processes are fixedly assigned a particular processor and thus cannot model processes being switched from one processor to another. Also, the semantics developed in [6, 7, 9, 18] inherently deal with only static systems and thus cannot capture most dynamical features of concurrent object-orientation such as dynamic creation of computational entities and dynamic sharing of communication handles.

There have been several timed extended process calculi in for example [2, 13, 15, 17, 23] for reasoning about real-time systems. However, their description is often at a too high abstraction level and thus does not always reflect on the details of programs for these systems. CSR [3] is a specification language whose semantics is defined by using a timed extended process calculus [2]. This approach is similar to ours, in which $\mathcal{R}_\mathcal{T}$ programs are translated into RtCCS. However, CSR cannot capture most dynamic features of concurrent object-orientation and cannot restrict the number of active processes.

In the remainder of this section we compare our work with some existing semantics for (non real-time) concurrent object-oriented programming languages based on process calculi. In [14, 20, 21], semantics for concurrent object-oriented programming languages are derived from a translation semantics for a parallel imperative programming language $\mathcal{M}$ into CCS expressions developed in [11], like ours. However, these semantics do not inherently have the notion of time and cannot specify the number of active objects which may execute simultaneously. In [14], semantics definition for concurrent object-oriented languages with inheritance is studied based on CCS but lacks any method for verifying programs. We believe that the way of modeling inheritance developed in [14] can easily be applied to our semantics definition. In [21] the author describes a semantics for POOL [1] based on π-calculus [12] which is an extension of CCS to accommodate port passing, whereas in [20] a semantics for a language like POOL based on a higher order process calculus which is an extension of CCS to accommodate process passing. However, in our approach such port or process passing is never required.

[8]There have been many temporal extensions of CSP, e.g., [4, 17] but they are rather true process calculi because of lacking high-level language constructions.

6 Conclusion

In this paper we developed a semantics for a real-time object-oriented programming language. The semantics is defined as a collection of translation rules from the syntactic constructions of the language into expressions in a timed extended process calculus, called RtCCS. By using the translation rules we can encode $\mathcal{R}_\mathcal{T}$'s arbitrary programs into expressions in RtCCS without losing any properties of the programs. Since RtCCS provides a theoretical framework to reason about real-time computation, we can strictly analyze and predict the temporal and behavioral properties of $\mathcal{R}_\mathcal{T}$'s programs through their corresponding expressions in RtCCS. Particularly, the semantics is unique in expressing the execution time and the upper bound of concurrent activities. It allows to analyze the temporal properties of programs executed with fewer processors than the number of active objects. Also, the approach presented in this paper is not essentially dependent on $\mathcal{R}_\mathcal{T}$ but is a general framework for reason about this kind of real-time languages with dynamical features such as runtime object (or process) creation, concurrency, and communication.

Finally, we point out our future works. We will attempt to define and compare various real-time languages through this semantics framework. Also, the semantics implicitly assumes a global clock, whereas in distributed systems such a global clock cannot inherently be realized. We have already developed a process calculus based on non global clocks in [16]. We plan to investigate a semantics for distributed real-time programming languages by using the calculus.

Acknowledgements

We are grateful to Vasco Vasconcelos (Keio University) for helpful comments on an earlier version of this paper. We would like to thank anonymous referees for providing many constructive and valuable suggestions.

References

[1] America, P., de Bakker, J., Kok, J., and Rutten, J., *Operational Semantics of a Parallel Object-Oriented Language*, Proceedings of ACM POPL'87, p194-208, 1987.

[2] Gerber, R. and Lee, I., *A Resource-Based Prioritized Bisimulation for Real-Time Systems*, Proceedings of CONCUR'90, October, LNCS 458, p263-277, Springer-Verlag, 1990.

[3] Gerber, R. and Lee, I., *A Layered Approach to Automating the Verification of Real-Time Systems*, IEEE Transaction on Software Engineering, Vol.18, No.9, p768-784, 1992.

[4] Gerth, R. and Boucher, A., *A Timed Failure Semantics for Extended Communicating Processes*, Proceedings of ICALP'87, LNCS 267, p95-114, Springer-Verlag, 1987.

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

[6] Hooman, J., *A Denotational Semantics for Shared Processors*, Proceeding of PARLE'91, LNCS 506, p184-201, Springer-Verlag, 1991.

[7] Huizing, C. Gerth, R. and deRover, W.P, *Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language*, Proceeding of ACM POPL'87, p223-237, 1987.

[8] Ishikawa, Y., Tokuda, H., and Mercer, C.W., *Object-Oriented Real-Time Language Design: Construction for Timing Constraints*, Proceeding of E-COOP/OOPSLA'90, 1990.

[9] Koymans, R. Shyamasundar, R.K, deRover, W.P, Gerth, R., and Arun-Kumer, S, *Compositional Semantics for Real-Time Distributed Computing*, Information and Computation, Vol.79, No.3, p210-256, 1988.

[10] Matsuoka, S., Taura, K., and Yonezawa, A., *Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages* Proceedings of ACM OOPSLA'93, p109-126, 1993.

[11] Milner, R. *Communication and Concurrency*, Prentice Hall, 1989.

[12] Milner, R., Parrow, J., and Walker, D., *A Calculus of Mobile Processes Part 1* & *2*, Technical report ECS-LFCS-89-85 & 86, University of Edinburgh, 1989.

[13] Nicollin. X., and Sifakis, j., *An Overview and Synthesis on Timed Process Algebras*, Proceedings of CAV'91, LNCS 575, p376-398, Springer-Verlag, 1991.

[14] Papathomas, M., *A Unifying Framework for Process Calculus Semantics of Concurrent Object-Based Languages and Features*, Proceeding of Workshop on Concurrent Object-Based Concurrent Computing, LNCS 612, p53-79, Springer-Verlag, 1992.

[15] Satoh, I., and Tokoro, M., *A Formalism for Real-Time Concurrent Object-Oriented Computing*, Proceedings of ACM OOPSLA'92, p315-326, 1992.

[16] Satoh, I., and Tokoro, M., *A Timed Calculus for Distributed Objects with Clocks*, Proceedings of E-COOP'93, LNCS 707, p326-345, Springer-Verlag, 1993.

[17] Schneider, S., Davies, J., Jackson, D.M., Reed, G.M., Reed, J.N., and Roscoe, A.W., *Timed CSP: Theory and Practice*, Proceedings of REX Workshop on Real-Time: Theory and Practice, LNCS 600, p640-675, Springer-Verlag, 1991.

[18] Shade, E., and Narayana, K.T., *Real-Time Semantics for Shared Variable Concurrency*, Information and Computation, Vol. 102, No.1, p56-82, 1993.

[19] Takashio, K., and Tokoro, M., *DROL: An Object-Oriented Programming Language for Distributed Real-time Systems*, Proceedings of ACM OOPSLA'92, p276-294, 1992.

[20] Thomsen, B., *Plain CHOCS: A Second Generation Calculus for Higher Order Processes*, Acta Informatica, Vol.30, No.1, p1-59, 1993.

[21] Walker, D., *π-Calculus Semantics of Object-Oriented Programming Languages*, Proceedings of Theoretical Aspects of Computer Software'91, LNCS 526, p532-547, Springer-Verlag, 1991.

[22] Wegner, P., *Concepts and Paradigms of Object-Oriented Programming*, ACM OOPS Messenger, Vol.1, No.1, August, 1990.

[23] Yi, W., *CCS + Time = an Interleaving Model for Real Time Systems*, Proceedings of ICALP'91, LNCS 510, p217-228, 1991.

[24] Yonezawa, A., and Tokoro, M., (ed), *Object-Oriented Concurrent Programming*, MIT Press, 1987.

```
program
  class Writer is
      var Y
    method start(X) in
      return true ;
      Y := X::put(1) timeout 3 ;

        ......

    endm
  endc
  class Reader is
      var Y
    method start(X) in
      return true ;
      Y := X::get(0) timeout 3 ;

        ......

    endm
  endc
  class Buffer  is
      var Y
    new
      Y := error
    method put(X) in
      Y := X ;
      return true
    endm
    method get(X) in
      return Y
    endm
  endc
is
  var B var R var W var X var Y
in
  B := Buffer::new ;
  W := Writer::new ;
  R := Reader::new ;
  X := W::start(B) timeout 5 ;
  Y := R::start(B) timeout 5
endp
```

Figure 11: Reader/Writer Program

Appendix

In this appendix we show a sketch of an analysis of a real-time program described in $\mathcal{R}_{\mathcal{T}}$. We consider a reader/writer problem program as shown in Figure 11. This program is constituted of an instance object of class `Buffer`, an instance object of class `Writer`, and an instance object of class `Reader`. The writer (reader) instance is invoked by a `start` message and then performs a `get` (`put`) method call with time constraint and then continues its own execution. Note that since in $\mathcal{R}_{\mathcal{T}}$ each instance contains at most one activity, the `put` method and `get` method of the buffer instance are never executed simultaneously.

By using the translation rules we will encode the program presented in Figure 11 into expressions of RtCCS as follows:

$$\begin{array}{rcl} [\![\textbf{program} \cdots \textbf{is} \cdots \textbf{endp}]\!] & = & \mathit{Prog} \;\; \text{where} \\ \mathit{Prog} & \overset{\text{def}}{=} & ([\![\textbf{class}\,\texttt{Buffer} \cdots \textbf{endc}]\!]| \\ & & \quad [\![\textbf{class}\,\texttt{Writer} \cdots \textbf{endc}]\!]| \\ & & \quad\; [\![\textbf{class}\,\texttt{Reader} \cdots \textbf{endc}]\!]| \\ & & \quad\;\; ([\![\textbf{var}\,\texttt{B} \cdots \textbf{var}\,\texttt{Y}]\!]\; \mathit{Before} \\ & & \quad\;\;\; [\![\texttt{B:=Buffer} :: \textbf{new}; \cdots \textbf{timeout}\,5]\!]) \\ & & \quad\;\;\;\; \mathit{ObjId}_0|\mathit{Machine}_N) \end{array}$$

where N is the number of available processors.

We introduce the following convenient notation to simplify the translation expression,

$$(t).P \overset{\text{def}}{=} \langle \mathbf{0}, P\rangle_t$$

where $(t).P$ means that the execution of P is suspended for t time units.

We first consider class `Buffer` declaration. The declaration is translated into the RtCCS expressions shown in Figure 12.

From the translated expression, we analyze the temporal properties of the instance of the class. For example, there are two $\mathsf{run}\cdots(t_{stat})\cdots\overline{\mathsf{idle}}$ sequences in "$\mathsf{new}_{Buffer} \cdots \mathit{Body}'_{Buffer,i}$" corresponding the initialization part of the instance. Hence, we know that the initialization time of an instance of `Buffer` is two t_{stat} time units. However, such a translated expression may be rather complex. Therefore, we formulate a useful method to reduce the complexity of the translated expressions, by using an equivalence relation.

Definition 1 Let us define $P \overset{\widehat{\mu}}{\Longrightarrow} P'$ as $P(\overset{\tau}{\longrightarrow})^* \overset{\mu}{\longrightarrow} (\overset{\tau}{\longrightarrow})^* P'$ if $\mu \neq \tau$ and if otherwise $P(\overset{\tau}{\longrightarrow})^* P'$. A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ is a *timed weak bisimulation* if $(P, Q) \in \mathcal{S}$ implies, for all $\mu \in \mathit{Act} \cup \{\surd\}$ is given by:

(i) Whenever $P \overset{\mu}{\longrightarrow} P'$ then, for some Q', $Q \overset{\widehat{\mu}}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{S}$.

(ii) Whenever $Q \overset{\mu}{\longrightarrow} Q'$ then, for some P', $P \overset{\widehat{\mu}}{\Longrightarrow} P'$ and $(P', Q') \in \mathcal{S}$.

$$
\begin{aligned}
&[\![\mathbf{class}\,\mathtt{Buffer}\ \mathbf{var}\ Y\ \cdots\mathbf{endc}]\!] \ = \ \mathit{ClassDec}_{\mathit{Buffer}} \quad \mathit{where}\\
\mathit{ClassDec}_{\mathit{Buffer}} &\stackrel{\mathrm{def}}{=} \mathsf{new}_{\mathit{Buffer}}.\mathsf{getid}(i).\overline{\mathsf{id}_{\mathit{Buffer}}}(i).(\mathit{ClassDec}_{\mathit{Buffer}}|\\
&\quad \mathsf{run}.((\mathit{Loc}_Y|\overline{\mathsf{done}}.\mathbf{0})[\mathsf{b}/\mathsf{done}]|\mathsf{b}.(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\\
&\quad \mathsf{run}.((\overline{\mathsf{res}}(\mathit{error}).\mathbf{0}|\mathsf{res}(x).\overline{\mathsf{write}_Y}(x).\overline{\mathsf{done}}.\mathbf{0})\setminus\{\mathsf{res}\}[\mathsf{b}/\mathsf{done}]|\mathsf{b}.(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\\
&\quad \mathit{Body}_{\mathit{Buffer},i})\setminus\{\mathsf{b}\})\setminus\{\mathsf{b}\}\setminus L_Y)\\
\mathit{Body}_{\mathit{Buffer},i} &\stackrel{\mathrm{def}}{=} (\mathsf{call}_{i,put}(x).(\mathsf{ans}(y).\overline{\mathsf{ret}_{i,put}}(y).\mathbf{0}|\mathit{Loc}_X|\overline{\mathsf{write}_X}(x).(\\
&\quad \mathsf{run}.((\mathsf{read}_X(z).\overline{\mathsf{res}}(z).\mathbf{0}|\mathsf{res}(u).\overline{\mathsf{write}_Y}(u).\overline{\mathsf{done}}.\mathbf{0})\setminus\{\mathsf{res}\}[\mathsf{b}/\mathsf{done}]|\mathsf{b}.(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\\
&\quad \mathsf{run}.((\overline{\mathsf{res}}(\mathit{true}).\mathbf{0}|\mathsf{res}(w).\overline{\mathsf{ans}}(w).\overline{\mathsf{done}}.\mathbf{0})\setminus\{\mathsf{res}\}[\mathsf{b}/\mathsf{done}]|\mathsf{b}.(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\\
&\quad \overline{\mathsf{endm}}.\mathbf{0})\setminus\{\mathsf{b}\})\setminus\{\mathsf{b}\}))\setminus L_X\setminus\{\mathsf{ans}\}\\
&\ \ + \mathsf{call}_{i,get}(x).(\mathsf{ans}(y).\overline{\mathsf{ret}_{i,get}}(y).\mathbf{0}|\mathit{Loc}_X|\overline{\mathsf{write}_X}(x).(\\
&\quad \mathsf{run}.((\mathsf{read}_Y(z).\overline{\mathsf{res}}(z).\mathbf{0}|\mathsf{res}(u).\overline{\mathsf{ans}}(u).\overline{\mathsf{done}}.\mathbf{0})\setminus\{\mathsf{res}\}[\mathsf{b}/\mathsf{done}]|\mathsf{b}.(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\\
&\quad \overline{\mathsf{endm}}.\mathbf{0})\setminus\{\mathsf{b}\}))\setminus L_X\setminus\{\mathsf{ans}\}\,|\,\mathsf{endm}.\mathit{Body}_{\mathit{Buffer},i})\setminus\{\mathsf{endm}\}
\end{aligned}
$$

Figure 12: Translation of `Buffer` class declaration

$$
\begin{aligned}
\mathit{ClassDec}_{\mathit{Buffer}} &\approx_{\mathcal{T}} \mathit{ClassDec}'_{\mathit{Buffer}}\\
\mathit{ClassDec}'_{\mathit{Buffer}} &\stackrel{\mathrm{def}}{=} \mathsf{new}_{\mathit{Buffer}}.\mathsf{getid}(i).\overline{\mathsf{id}_{\mathit{Buffer}}}(i).(\mathit{ClassDec}'_{\mathit{Buffer}}|\mathsf{run}.(\mathit{Loc}_Y|(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\\
&\quad \mathsf{run}.\overline{\mathsf{write}_Y}(\mathit{error}).(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\mathit{Body}'_{\mathit{Buffer},i})\setminus L_Y)\\
\mathit{Body}'_{\mathit{Buffer},i} &\stackrel{\mathrm{def}}{=} \mathsf{call}_{i,put}(x).\mathsf{run}.\overline{\mathsf{write}_Y}(x).(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\\
&\qquad\qquad \mathsf{run}.\overline{\mathsf{ret}_{i,put}}(\mathit{true}).(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\mathit{Body}'_{\mathit{Buffer},i}\\
&\ \ + \mathsf{call}_{i,get}(x).\mathsf{run}.\mathsf{read}_Y(y).\overline{\mathsf{ret}_{i,put}}(y).(t_{\mathit{stat}}).\overline{\mathsf{idle}}.\mathit{Body}'_{\mathit{Buffer},i}
\end{aligned}
$$

Figure 13: Equivalent expression of `Buffer` class declaration

Let "$\approx_{\mathcal{T}}$" denote the largest timed weak bisimulation, and call P and Q *timed observation equivalent* if $P \approx_{\mathcal{T}} Q$.

Intuitively, if P and Q are timed observation equivalent, each action of P must be matched by a sequence of actions of Q with the same observable contents and timings, and conversely.

The equivalence can equate objects (processes) that are not distinguishable by the observable behavior and the timing of their computations. Especially, encapsulation in object-orientation prevents the internal computation of an instance object from interacting with its external environment, vice versa. Consequently, the above timed equivalence is appropriate and practical to equate two objects whose internal implementations are different from one another. We leave the further details on the equivalence to [15].

By using the equivalence, $\mathit{ClassDec}_{\mathit{Buffer}}$ can be transformed into an equivalent process $\mathit{ClassDec}'_{\mathit{Buffer}}$ that has substantially less complexity in its structure as given in Figure 13. Since the simplified expression completely coincides with its original one in their external behaviors and timings, we can easily analyze the external properties of the class declaration through the simplified expression.

Also, we should emphasize that the timed observation equivalence provides a powerful method to verify real-time programs. For example, let $\mathit{ClassDec}'_{\mathit{Buffer}}$ be a specification of the buffer class and let $\mathit{ClassDec}_{\mathit{Buffer}}$ be an implementation of the class. The result in Figure 13 shows that the implementation satisfies the external behavioral requirements and their timing ones in the specification.

By using a similar way, we can reduce the complexity of the other translated expressions as given in Figure 14.

We present notable algebraic properties of the timed observation equivalence.

Proposition 2 Let $P_1 \approx_{\mathcal{T}} P_2$. Then

(1) $\alpha.P_1 \approx_{\mathcal{T}} \alpha.P_2$ (2) $P_1|Q \approx_{\mathcal{T}} P_2|Q$
(3) $P_1 \setminus L \approx_{\mathcal{T}} P_2 \setminus L$ (4) $P_1[f] \approx_{\mathcal{T}} P_2[f]$
(5) $\langle Q, P_1\rangle_t \approx_{\mathcal{T}} \langle Q, P_2\rangle_t$

The above proposition shows that the timed observation equivalence is preserved by all operators that are used to combine RtCCS expressions translated from $\mathcal{R}_{\mathcal{T}}$'s class declarations, statements, and expressions. Therefore, we guarantee that the above simplified expressions are substitutable for their own original expressions.

$$\begin{array}{rcl}
[\![\textbf{class}\,\texttt{Writer}\ \textbf{var}\ Y\ \cdots\textbf{endc}]\!] & \approx_{\mathcal{T}} & \mathit{ClassDec}'_{Writer} \quad \text{where} \\
\mathit{ClassDec}'_{Writer} & \stackrel{\text{def}}{=} & \mathsf{new}_{Writer}.\mathsf{getid}(i).\overline{\mathsf{id}_{Writer}}(i).(\mathit{ClassDec}'_{Writer}| \\
 & & \mathsf{run}.(\mathit{Loc}_Y\,|(t_{stat}).\overline{\mathsf{idle}}.\mathit{Body}'_{Writer,i})) \setminus L_Y \\
\mathit{Body}'_{Writer,i} & \stackrel{\text{def}}{=} & \mathsf{call}_{i,start}(j).(\mathit{Loc}_X\,|\overline{\mathsf{write}_X}(j).\mathsf{run}.\overline{\mathsf{ret}_{i,start}}(true).(t_{stat}).\overline{\mathsf{idle}}. \\
 & & \mathsf{run}.(\langle\overline{\mathsf{call}_{j,put}}(\hat{1}).\mathsf{ret}_{j,put}(x).\overline{\mathsf{reply}}(x).\mathbf{0}\,,\ \overline{\mathsf{abort}}.\mathbf{0}\rangle_3\,| \\
 & & \quad\langle\mathsf{reply}(y).\overline{\mathsf{write}_Y}(y).(t_{stat}).\overline{\mathsf{idle}}.\cdots\cdots\mathit{Body}'_{Writer,i}\,,\ \mathsf{reply}(y).\mathbf{0}+\mathsf{abort}.\mathbf{0}\,| \\
 & & \quad\overline{\mathsf{write}_Y}(error).(t_{stat}).\overline{\mathsf{idle}}.\cdots\cdots\mathit{Body}'_{Writer,i}\rangle_3) \setminus \{\mathsf{reply},\mathsf{abort}\}) \setminus L_X
\end{array}$$

$$\begin{array}{rcl}
[\![\textbf{class}\,\texttt{Reader}\ \textbf{var}\ Y\ \cdots\textbf{endc}]\!] & \approx_{\mathcal{T}} & \mathit{ClassDec}'_{Reader} \quad \text{where} \\
\mathit{ClassDec}'_{Reader} & \stackrel{\text{def}}{=} & \mathsf{new}_{Reader}.\mathsf{getid}(i).\overline{\mathsf{id}_{Reader}}(i).(\mathit{ClassDec}'_{Reader}| \\
 & & \mathsf{run}.(\mathit{Loc}_Y\,|(t_{stat}).\overline{\mathsf{idle}}.\mathit{Body}'_{Reader,i})) \setminus L_Y \\
\mathit{Body}'_{Reader,i} & \stackrel{\text{def}}{=} & \mathsf{call}_{i,start}(j).(\mathit{Loc}_X\,|\overline{\mathsf{write}_X}(j).\mathsf{run}.\overline{\mathsf{ret}_{i,start}}(true).(t_{stat}).\overline{\mathsf{idle}}. \\
 & & \mathsf{run}.(\langle\overline{\mathsf{call}_{j,get}}(\hat{0}).\mathsf{ret}_{j,get}(x).\overline{\mathsf{reply}}(x).\mathbf{0}\,,\ \overline{\mathsf{abort}}.\mathbf{0}\rangle_3\,| \\
 & & \quad\langle\mathsf{reply}(y).\overline{\mathsf{write}_Y}(y).(t_{stat}).\overline{\mathsf{idle}}.\cdots\cdots\mathit{Body}'_{Reader,i}\,,\ \mathsf{reply}(y).\mathbf{0}+\mathsf{abort}.\mathbf{0}\,| \\
 & & \quad\overline{\mathsf{write}_Y}(error).(t_{stat}).\overline{\mathsf{idle}}.\cdots\cdots\mathit{Body}'_{Reader,i}\rangle_3) \setminus \{\mathsf{reply},\mathsf{abort}\}) \setminus L_X
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{l}{[\![\textbf{var}\,\texttt{B}\ \textbf{var}\,\texttt{W}\ \textbf{var}\,\texttt{R}\ \textbf{var}\,\texttt{X}\ \textbf{var}\,\texttt{Y}]\!]\ \mathit{Before}\ [\![\texttt{B:=Buffer}::\textbf{new};\cdots\texttt{Y:=R::start(B)}\ \textbf{timeout}\ 5]\!] \ \approx_{\mathcal{T}}\ \mathit{Initial}'} \\
\text{where}\quad \mathit{Initial}' & \stackrel{\text{def}}{=} & (\mathsf{run}.(t_{stat}).\overline{\mathsf{idle}}.)^5 \\
 & & \mathsf{run}.\overline{\mathsf{new}_{Buffer}}.\mathsf{id}_{Buffer}(i).(t_{stat}).\overline{\mathsf{idle}}. \\
 & & \mathsf{run}.\overline{\mathsf{new}_{Writer}}.\mathsf{id}_{Writer}(j).(t_{stat}).\overline{\mathsf{idle}}. \\
 & & \mathsf{run}.\overline{\mathsf{new}_{Reader}}.\mathsf{id}_{Reader}(k).(t_{stat}).\overline{\mathsf{idle}}.\,A \\
A & \stackrel{\text{def}}{=} & \mathsf{run}.(\langle\overline{\mathsf{call}_{j,start}}(i).\mathsf{ret}_{j,start}(x).\overline{\mathsf{reply}}(x).\mathbf{0}\,,\ \overline{\mathsf{abort}}.\mathbf{0}\rangle_5\,| \\
 & & \quad\langle\mathsf{reply}(y).(t_{stat}).\overline{\mathsf{idle}}.B\,,\ \mathsf{reply}(y).\mathbf{0}+\mathsf{abort}.\mathbf{0}\,|\,(t_{stat}).\overline{\mathsf{idle}}.B\rangle_5) \setminus \{\mathsf{reply},\mathsf{abort}\} \\
B & \stackrel{\text{def}}{=} & \mathsf{run}.(\langle\overline{\mathsf{call}_{k,start}}(i).\mathsf{ret}_{k,start}(x).\overline{\mathsf{reply}}(x).\mathbf{0}\,,\ \overline{\mathsf{abort}}.\mathbf{0}\rangle_5\,| \\
 & & \quad\langle\mathsf{reply}(y).(t_{stat}).\overline{\mathsf{idle}}.\mathbf{0}\,,\ \mathsf{reply}(y).\mathbf{0}+\mathsf{abort}.\mathbf{0}\,|\,(t_{stat}).\overline{\mathsf{idle}}.\mathbf{0}\rangle_5) \setminus \{\mathsf{reply},\mathsf{abort}\}
\end{array}$$

Figure 14: Equivalent expressions of `Reader`, `Write` class and initial program

The entire program can be constructed by a parallel composition of the simplified expressions as follow:

$$\begin{array}{rcl}
\mathit{Prog} & \approx_{\mathcal{T}} & \mathit{Prog}' \quad \text{where} \\
\mathit{Prog}' & \stackrel{\text{def}}{=} & (\mathit{ClassDec}'_{Buffer}|\mathit{ClassDec}'_{Writer}| \\
 & & \quad \mathit{ClassDec}'_{Reader}|\mathit{ObjId}_0| \\
 & & \quad\quad \mathit{Machine}_N|\mathit{Initial}')
\end{array}$$

By expanding the above simplified expressions[9], we can analyze both the behavioral properties of the entire program and its temporal properties through the theoretical framework of RtCCS. Particularly, the upper bound of available processors can be specified as N in $\mathit{Machine}_N$ and thus we can predict the execution of programs with a given number of processors. From the above expression we know that, for example, let $N = 2$, then `put` (or `get`) method call may occasionally timeout and the execution of the program described in Figure 11 takes from 14 t_{stat} to 16 t_{stat} time units. The final content of variable `Y` in `Writer`'s instance and that of `Reader`'s one are non deterministic. Let $N \geq 3$, then no method calls timeouts and the execution of the program takes from 12 t_{stat} to 13 t_{stat} time units ($N = 3$), or 12 t_{stat} time units ($N \geq 4$). The final content of variable `Y` in `Writer`'s instance and that of `Reader`'s one are deterministic (**`true`** and `1` respectively).

[9]We omit the expansion and the detail analysis for lack of space.