Modeling and Verification of Distributed Real-Time Systems Based on CafeOBJ

Kazuhiro Ogata, Kokichi Futatsugi
Graduate School of Information Science, JAIST
Tatsunokuchi, Ishikawa 923-1292, Japan
{ogata, kokichi}@jaist.ac.jp

Abstract

CafeOBJ is a wide spectrum formal specification language based on multiple logical foundations: mainly initial and hidden algebra. A wide range of systems can be specified in CafeOBJ thanks to its multiple logical foundations. However, distributed real-time systems happen to be excluded from targets of CafeOBJ. In this paper, we propose a method of modeling and verifying such systems based on CafeOBJ, together with timed evolution of UNITY computational models.

Keywords: algebraic specification, CafeOBJ, distributed real-time systems, modeling, specification, UNITY, verification.

1. Introduction

CafeOBJ[2, 4] is a wide spectrum formal specification language based on multiple logical foundations: mainly initial and hidden algebra. A wide range of systems can be specified in CafeOBJ thanks to its multiple logical foundations. Not only are static aspects of systems specified in terms of initial algebra, but dynamic aspects of systems are also specified in terms of hidden algebra.

Besides, CafeOBJ can be applied to distributed systems. We have done some case studies[10, 11, 12]. In the case studies, distributed systems are modeled as UNITY[3] computational models and the models are described in CafeOBJ. Since the CafeOBJ system can be used as an interactive theorem prover, we have verified that the distributed systems have some desirable properties with the help of the CafeOBJ system.

However, distributed real-time systems happen to be excluded from targets of CafeOBJ. Most of the crucial systems fielded in the real world are distributed real-time systems such as flight control systems and patient monitoring systems. Hence it is very important to include such systems as targets of CafeOBJ. On the other hand, distributed real-time systems would be specified from various points of view in CafeOBJ, which we believe is also very meaningful.

In this paper, we propose a method of modeling and verifying distributed real-time systems based on CafeOBJ, together with timed evolution of UNITY computational models called TBCMs. In the method, such systems are modeled as TBCMs, the models are described in CafeOBJ, and it is verified that the systems have some desirable properties with the help of the CafeOBJ systems. Two examples are used to present the method.

The rest of the paper is organized as follows. In Sect. 2, UNITY computational models are reformulated in the same manner as the definition of fair transition systems[9]. The reformulated ones are called BCMs. We also describe how to specify BCMs in CafeOBJ using an example (asynchronous data sending problem). In Sect. 3, timed evolution of BCMs called TBCMs is described. The example constrained by timing is modeled as a TBCM, the TBCM is described in CafeOBJ, and it is verified that the system has a property with the help of the CafeOBJ system. Section 4 describes another example (Fischer's protocol). Section 5 finally concludes the paper.

2. Description of distributed systems in CafeOBJ

2.1. Basic computational models: transition systems

UNITY computational models are reformulated in the same manner as the definition of fair transition systems [9]. The reformulated ones are called basic computational models, or BCMs. A BCM $\mathcal{S} = \langle \mathcal{V}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{V}$: A set of variables. The variables (or their possible values) form the state space Σ of $\mathcal{S}$, and a state of $\mathcal{S}$ is a point, or an element of Σ.
- $\mathcal{I}$: The initial condition. It specifies the initial values of the variables. Since some variables may not be specified by $\mathcal{I}$, $\mathcal{S}$ may have more than one initial state.

- $\mathcal{T}$: A set of transition rules. Each transition rule $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \to \Sigma$ mapping each state $s \in \Sigma$ into a successor state $\tau(s) \in \Sigma$. Transition rules are generally defined together with conditions on which the transition rules are *effectively* executed. If the condition of a transition rule is false in a state of $\mathcal{S}$, namely that the transition rule is not *effective* in the state, its execution does not change the state at all.

An execution starts from one initial state and goes on forever; in each step of execution some transition rule is selected nondeterministically and executed. Nondeterministic selection is constrained by the following fairness rule: every transition rule is selected infinitely often. Given a BCM, a set of infinite sequences of states is obtained from execution, constrained by the fairness rule, of the BCM. Such an infinite sequence of states is called a computation of the BCM. More specifically, a computation of a BCM $\mathcal{S}$ is an infinite sequence $s_0, s_1, \ldots$ of states satisfying:

- *Initiation* : For each $v \in \mathcal{V}$, v satisfies $\mathcal{I}$ in s_0.
- *Consecution* : For each $i \in \{0, 1, \ldots\}$, $s_{i+1} = \tau(s_i)$ for some $\tau \in \mathcal{T}$.
- *Fairness* : For each $\tau \in \mathcal{T}$, there exist infinite number of indexes $i \in \{0, 1, \ldots\}$ such that $s_{i+1} = \tau(s_i)$.

A state of $\mathcal{S}$ is called *reachable* if it appears in a computation of $\mathcal{S}$.

2.2. How to specify distributed systems in CafeOBJ

CafeOBJ[2, 4] is mainly based on two logical foundations: *initial* and *hidden* algebra. Initial algebra is used to specify abstract data types, and hidden algebra[5] to specify objects in object-orientation. There are two kinds of sorts in CafeOBJ: *visible* and *hidden* sorts. A visible sort represents an abstract data type, and a hidden sort the state space of an object. There are basically two kinds of operations to hidden sorts: *action* and *observation* operations. An action operation, or an action can change a state of an object. An observation operation, or an observation can be used to observe the value of a data component in an object. Both actions and observations are defined with equations.

Declarations of visible sorts are enclosed with `[` and `]`, and those of hidden ones with `*[` and `]*`. Declarations of observations and actions start with `bop` or `bops`, and those of other operations with `op` or `ops`. After `bop` or `op` (or `bops` or `ops`), an operator is written (or more than one operator is written), followed by `:` and a sequence of sorts, and ended with `->` and one sort. Definitions of equations start with `eq`, and those of conditional ones with `ceq`. After `eq`, two terms connected by `=` are written, and ended with a full stop. After `ceq`, two terms connected by `=` are written, followed by `if` and a term denoting a condition, and ended with a full stop.

Since objects can be regarded as transition systems, a BCM can be naturally described in CafeOBJ. The state space of a BCM is denoted by a hidden sort. A single variable or a set of variables is represented by an observation. Given a BCM in which a set $\{x_i \mid i \in \{1, 2, \ldots\}\}$ of variables which types are natural numbers is used, the variables are represented by an observation declared as `bop x : Sys NzNat -> Nat`, where `Sys` is the hidden sort denoting the state space of the BCM, and `NzNat` and `Nat` are the visible sorts denoting non-zero natural numbers and natural numbers respectively. The value of x_i in a state s is denoted by `x(`s`,`i`)`. The initial values of variables are specified with equations. An operation denoting any initial state is first declared as `op init : -> Sys .` Then if all x_i's are initially 0, we have the equation `eq x(init,I) = 0 .`, where `I` is a CafeOBJ variable whose sort is `NzNat`.

A single transition rule or a set of transition rules is represented by an action. Given a BCM in which a set $\{t_i \mid i \in \{1, 2, \ldots\}\}$ of transition rules is used, the transition rules are represented by an action declared as `bop t : Sys NzNat -> Sys .` The successor state after executing t_i in a state s is denoted by `t(`s`,`i`)`. The behavior of transition rules is specified with equations that define how the state (namely the variables) changes if each transition rule is executed in a state. Suppose that if t_i is executed, x_i is incremented and x_j $(j \neq i)$ is left unchanged. Then, we have two equations `eq x(t(S,I),I) = x(S,I) + 1 .` and `ceq x(t(S,I),J) = x(S,J) if I =/= J .` , where `S`, `I` and `J` are CafeOBJ variables whose sorts are `Sys`, `NzNat` and `NzNat` respectively.

2.3 Example: asynchronous data sending problem

As an example, let us consider an *asynchronous data sending problem*: a sender repeatedly sends natural numbers one by one from zero in increasing order to a receiver via a cell. The sender puts a natural number into the cell, and the receiver gets the natural number from the cell if the cell is not empty.

We first model this system as a BCM. A number of variables and transition rules depend on which level of detail the system gets specified at. Here we use four variables and two transition rules to model the system. The four variables are *empty*, *content*, *data*, and *list*. The cell is represented by *empty* and *content*. If the cell is empty, (the value of) *empty* is true, and otherwise *empty* is false and *content* is the natural number in the cell. The natural number sent next by the sender is represented by *data*. If the receiver gets a natural number from the cell, it puts the number into a list at

the end. The list is represented by *list* that is mainly needed to verify that the system constrained by timing has a safety property (described later). The initial values of *empty*, *content*, *data*, and *list* are true, unspecified, zero, and nil. The two transition rules are *send* and *rec* that correspond to that the sender puts a natural number into the cell and that the receiver gets a natural number from the cell, respectively. *send* is always effective, and *rec* is effective if and only if the cell is non-empty.

We then describe the BCM modeling the system in CafeOBJ. The main part of the signature is as follows:

```
*[Sys]*
-- any initial state
op   init         :     -> Sys
-- observations
bop  empty        : Sys -> Bool
bops content data : Sys -> Nat
bop  list         : Sys -> List
-- actions
bops send rec     : Sys -> Sys
```

A comment starts with `--` and terminates at the end of the line. Hidden sort `Sys` represents the state space Σ of the BCM, and visible sorts `Bool` and `List` represent booleans and lists of natural numbers, respectively. `init` denotes any initial state of the BCM.

We have three sets of equations in the specification: one for any initial state, and the others for the two actions.

The equations defining any initial state are as follows:

```
eq empty(init) = true .
eq data(init)  = 0 .
eq list(init)  = nil .
```

Since the cell is initially empty, the initial value of `content` is not specified, namely that we have no equation for `content` in any initial state.

The equations defining how a state of the BCM changes if transition rule *send* is executed in the state are as follows:

```
eq empty(send(S))   = false .
eq content(send(S)) = data(S) .
eq data(send(S))    = inc(data(S)) .
eq list(send(S))    = list(S) .
```

where `S` is a CafeOBJ variable whose sort is `Sys`. These equations indicate that if *send* is executed in a state, then *empty* is set to false, *content* is set to the old value of *data*, *data* is incremented, and *list* is left unchanged.

The equations defining how a state of the BCM changes if transition rule *rec* is executed in the state are as follows:

```
ceq empty(rec(S))   = true
    if empty(S) == false .
ceq empty(rec(S))   = empty(S)
    if empty(S) == true .
eq  content(rec(S)) = content(S) .
eq  data(rec(S))    = data(S) .
ceq list(rec(S))    = put(list(S),content(S))
    if empty(S) == false .
ceq list(rec(S))    = list(S)
    if empty(S) == true .
```

In conditions of conditional equations, `==` is used in lieu of `=` for equality. In a state where *empty* is false, if *rec* is executed, then *empty* is set to true and the old value of *content* is put into *list* at the end. In a state where *empty* is true, even if *rec* is executed, nothing changes. In any state, *content* and *data* are left unchanged even if *rec* is executed.

In the system specified now, the receiver may not receive all the natural numbers sent by the sender. A computation corresponding to the case in which the receiver only receives odd numbers is as follows: `init`, `send(init)`, `send(send(init))`, `rec(send(send(init)))`, ...

3. Description of distributed real-time systems in CafeOBJ

3.1. Timed evolution of basic computational models

By introducing clock variables, BCMs are evolved into computational models that can deal with timing. The computational models are called timed evolution of basic computational models, or TBCMs. A TBCM $\mathcal{S} = \langle \mathcal{V}, \mathcal{I}, \mathcal{T} \cup \{tick_r \mid r \in R^+\} \rangle$ consists of:

- $\mathcal{V}$: A set of variables as a BCM. But, the set $\mathcal{V} = \mathcal{D} \cup \mathcal{C}$ is classified into the set $\mathcal{D}$ of *discrete variables* and the set $\mathcal{C}$ of *clock variables*. The types of clock variables are non-negative real numbers (R^+) or infinity (∞). For each $\tau \in \mathcal{T}$, there are two clock variables $l_\tau : R^+$ and $u_\tau : (R^+ \setminus \{0\}) \cup \{\infty\}$ that are called the *lower* and *upper bounds* of τ. They are basically used to force τ to be executed between l_τ and u_τ. Besides, there is one special clock $now : R^+$. It serves as the master clock and always indicates the elapse of time after a system starts its computation.

- $\mathcal{I}$: The initial condition as a BCM. Master clock *now* is initially set to 0.

- $\mathcal{T} \cup \{tick_r \mid r \in R^+\}$: A set of transition rules as a BCM. But, there is a set of time advancing transition rules $tick_r : \Sigma \to \Sigma$. It advances *now* by r unless $now + r$ exceeds upper bound u_τ for any $\tau \in \mathcal{T}$. $tick_r$ does not change any variable except *now*, and any transition rule except $tick_r$ does not affect advancing *now*. For each $\tau \in \mathcal{T}$, in addition to the condition on which τ gets effective, $l_\tau \leq now$ is given to τ as a conjunct of its condition.

For each $\tau \in \mathcal{T}$, besides two clock variables l_τ and u_τ, there are two constants d_τ^{min} and d_τ^{max} whose types are the same as l_τ and u_τ, respectively. d_τ^{min} and d_τ^{max} are called the *minimum* and *maximum delays* of τ. Basically when τ newly gets effective, τ must be executed between d_τ^{min} and

d_τ^{max} time units later from that time unless τ gets ineffective meanwhile. If τ is initially effective, l_τ and u_τ are initially set to d_τ^{min} and d_τ^{max}, respectively, and otherwise l_τ and u_τ are initially set to 0 and ∞, respectively. Suppose that τ is executed in a state s in which it is effective and $l_\tau \leq now$, and let s' be the successor state. If τ is still effective in s', l_τ and u_τ are set to $now + d_\tau^{min}$ and $now + d_\tau^{max}$, respectively, and otherwise l_τ and u_τ are set to 0 and ∞, respectively. For any other transition rule $\tau' \in \mathcal{T}$, if it is ineffective (or effective) in s and it gets effective (or ineffective) in s', then $l_{\tau'}$ and $u_{\tau'}$ are set to $now + d_{\tau'}^{min}$ (or 0) and $now + d_{\tau'}^{max}$ (or ∞), respectively. Otherwise, $l_{\tau'}$ and $u_{\tau'}$ are left unchanged.

A computation of a TBCM $\mathcal{S}$ is an infinite sequence $s_0, s_1, \ldots, s_i, \ldots$ of states satisfying *Initiation* and *Consecution* as a BCM, but satisfying *Time Divergence* instead of *Fairness*.

- *Time Divergence*: As i increases, *now* increases without bound.

A TBCM is defined to be *non-Zeno* if any finite sequence of states generated by the TBCM can be extended to a computation. A sufficient condition on which a TBCM $\mathcal{S}$ is non-Zeno is that for each $\tau \in \mathcal{T}$, always $l_\tau \leq u_\tau$, namely $d_\tau^{min} \leq d_\tau^{max}$ (from [1]). If for some $\tau \in \mathcal{T}$, d_τ^{min} (or d_τ^{max}) is 0 (or ∞), then l_τ (or u_τ) may be omitted from $\mathcal{V}$.

In the verification given in the remainder of the paper, we use the following lemma (from [8]):

Lemma 1. *In any reachable state s of a* TBCM *$\mathcal{S}$, the following hold:*

- *$now \leq u_\tau$ for each $\tau \in \mathcal{T}$.*
- *If $\tau \in \mathcal{T}$ is effective, then $u_\tau \leq now + d_\tau^{max}$.*

3.2. How to specify distributed real-time systems in CafeOBJ

Let us consider the asynchronous data sending problem again. By giving lower and upper bounds to some transition rules, the system is made reliable, namely that no natural number sent by the sender is lost.

Basically the system is modeled as before. In addition to the four variables and two transition rules, however, five clock variables and a set of time advancing transition rules $\{tick_r \mid r \in R^+\}$ are added. One of the clock variables is *now*. The remaining four clock variables are l_{send}, u_{send}, l_{rec}, and u_{rec}. Besides, we have four constants d_{send}^{min}, d_{send}^{max}, d_{rec}^{min}, and d_{rec}^{max}. We decide the values of some of the constants and their relations as follows: $d_{send}^{max} = \infty$, $d_{rec}^{min} = 0$, and $0 < d_{rec}^{max} < d_{send}^{min} < \infty$. As mentioned before, u_{send} and l_{rec} are omitted. The timing constraints propose that the receiver should get the natural number from the cell before the sender puts a new natural number into the cell if the cell is not empty, or the sender should not put a new natural number into the cell before the receiver gets the natural number from the cell if the cell is not empty.

We describe the TBCM modeling the system constrained by timing in CafeOBJ. Specifying a TBCM in CafeOBJ is basically the same as specifying a BCM in CafeOBJ.

The main part of the signature is as follows:

```
*[Sys]*
-- any initial state
op   init         :           -> Sys
-- observations
bop  empty        : Sys       -> Bool
bops content data : Sys       -> Nat
bop  list         : Sys       -> List
bops now l        : Sys       -> Real+
bop  u            : Sys       -> Timeval
-- actions
bops send rec     : Sys       -> Sys
bop  tick         : Sys Real+ -> Sys
-- delays
ops  d1 d2        :           -> Real+
```

Visible sorts `Real+` and `Timeval` represent R^+ and $(R^+ \setminus \{0\}) \cup \{\infty\}$, respectively. Observations `now`, `l`, and `u` correspond to variables *now*, l_{send}, and u_{rec}, respectively, and action `tick` to $\{tick_r \mid r \in R^+\}$. `d1` and `d2` correspond to d_{send}^{min} and d_{rec}^{max}, respectively.

We have four sets of equations in the specification: one for any initial state, and the others for the three actions. The equations defining any initial state are as follows:

```
eq empty(init) = true .
eq data(init)  = 0 .
eq list(init)  = nil .
eq now(init)   = 0 .
eq l(init)     = d1 .
eq u(init)     = oo .
```

`oo` stands for ∞. Initially l_{send} (or `l`) is set to d_{send}^{min} (or `d1`) because *send* is effective, while u_{rec} (or `u`) is set to ∞ (or `oo`) because *rec* is not effective.

The equations defining how a state of the TBCM changes if *send* is executed in the state are as follows:

```
ceq empty(send(S))   = false if l(S) <= now(S) .
ceq empty(send(S))   = empty(S) if now(S) < l(S) .
ceq content(send(S)) = data(S) if l(S) <= now(S) .
ceq content(send(S)) = content(S)
    if now(S) < l(S) .
ceq data(send(S))    = inc(data(S))
    if l(S) <= now(S) .
ceq data(send(S))    = data(S) if now(S) < l(S) .
eq  list(send(S))    = list(S) .
eq  now(send(S))     = now(S) .
ceq l(send(S))       = now(S) + d1
    if l(S) <= now(S) .
ceq l(send(S))       = l(S) if now(S) < l(S) .
ceq u(send(S))       = now(S) + d2
    if l(S) <= now(S) .
ceq u(send(S))       = u(S) if now(S) < l(S) .
```

If $now < l_{send}$, execution of *send* does not change the TBCM at all. Otherwise, four discrete variables change as described before, and l_{send} and u_{rec} are set to $now + d_{send}^{min}$ and $now + d_{rec}^{max}$, respectively, for the reason stated above.

The equations defining how a state of the TBCM changes if *rec* is executed in the state are as follows:

```
ceq empty(rec(S))   = true if empty(S) == false .
ceq empty(rec(S))   = empty(S)
    if empty(S) == true .
eq  content(rec(S)) = content(S) .
eq  data(rec(S))    = data(S) .
ceq list(rec(S))    = put(list(S),content(S))
    if empty(S) == false .
ceq list(rec(S))    = list(S) if empty(S) == true .
eq  now(rec(S))     = now(S) .
eq  l(rec(S))       = l(S) .
ceq u(rec(S))       = oo if empty(S) == false .
ceq u(rec(S))       = u(S) if empty(S) == true .
```

If *empty* is true, execution of *rec* does not change the TBCM at all. Otherwise, four discrete variables change as described before, and l_{send} and u_{rec} are left unchanged and set to ∞, respectively, for the reason stated above.

The equations defining how a state of the TBCM changes if $tick_{\mathtt{D}}$ ($\mathtt{D} \in R^+$) is executed in the state are as follows:

```
eq  empty(tick(S,D))   = empty(S) .
eq  content(tick(S,D)) = content(S) .
eq  data(tick(S,D))    = data(S) .
eq  list(tick(S,D))    = list(S) .
ceq now(tick(S,D))     = now(S) + D
    if now(S) + D <= u(S) .
ceq now(tick(S,D))     = now(S)
    if u(S) < now(S) + D .
eq  l(tick(S,D))       = l(S) .
eq  u(tick(S,D))       = u(S) .
```

now gets advanced but does not go beyond u_{rec}.

Note that we have another equation "`eq d2 < d1 = true .`" because $d_{rec}^{max} < d_{send}^{min}$.

3.3. How to verify distributed real-time systems with CafeOBJ

Properties to prove in this paper are safety ones. If a property holds in any reachable state of a TBCM, the property is called safety and the TBCM has the safety property. Since any transition rule does not break safety properties, safety properties are often interpreted as saying that some particular bad thing never happens. Proofs that a TBCM has a safety property can be done by structural induction on transition rules. We first confirm that the property candidate holds in any initial state, and then, that the property candidate is preserved by every transition rule. The CafeOBJ system can be used as an interactive theorem prover, and such proofs can be done with the help of the CafeOBJ system.

We prove that the system that has just been specified has the safety property that no natural number sent by the sender is lost. More precisely, the following claim is shown.

Claim 1. *In any reachable state, list equals* $[0..data - 1]$ *if empty is true, and* $[0..data - 2]$ *if empty is false, where* $[m..n]$ *is the list of natural numbers from* m *through* n *if* $m \leq n$*, and nil if* $m > n$*.*

Proof. The claim is true in any initial state because *empty* is true, *data* is 0 and *list* is nil. Then, the claim is shown to be preserved by every transition rule. Suppose that the claim holds in a state s, we show that it still holds in the successor state s' after any transition rule is executed in s.

First we show that *send* preserves the claim. The case is divided into two sub-cases: 1) *empty* is true, and 2) *empty* is false. Since execution of *send* does not change the system at all if $now < l_{send}$ (if so, *send* clearly preserves the claim), we only consider the case in which $l_{send} \leq now$.

In case (1), the following proof score can be used to show that *send* preserves the claim.

```
open ASEND
  ops s s' : -> Sys .
  op  n    : -> Nat .
  eq  data(s)        = n .
  eq  empty(s)       = true . -- assumption.
  eq  l(s) <= now(s) = true . -- assumption.
  eq  s'             = send(s) .
  red data(s')  == s(n) and empty(s') == false
      and list(s')  == list(s) .
close
```

ASEND is the name of the module in which the specification of the system is written. By opening the module using CafeOBJ command `open`, the definitions in the specification can be used. `s` of `s(n)` is the successor function of natural numbers. The term following CafeOBJ command `red`(`uction`) means that the latter case of the claim holds in s'. `red` reduces the term by regarding equations as left-to-right rewrite rules. In this case, the term is reduced to true, which implies that *send* preserves the claim.

In case (2), we use another claim that if *empty* is false, then $u_{rec} < l_{send}$ due to $d_{rec}^{max} < d_{send}^{min}$, which can be easily proven. From this inequality and Lemma 1 (namely $now \leq u_{rec}$), $now < l_{send}$, which contradicts the assumption that $l_{send} \leq now$.

Second we show that *rec* preserves the claim. The case is divided into two sub-cases (1) and (2) as well. In case (1), *rec* is not effective. In case (2), we use another claim that if *empty* is false, then $data > 0$ and $content = data - 1$, which can be easily proven as well. The following proof score can be used to show that *rec* preserves the claim.

```
open ASEND
  ops s s' : -> Sys .
  op  n    : -> Nat .
  eq  empty(s)   = false .  -- assumption.
  eq  content(s) = n .      -- another claim.
  eq  data(s)    = s(n) .   -- another claim.
  eq  s' = rec(s) .
  red data(s')  == s(n) and empty(s') == true
      and list(s')  == put(list(s),n) .
close
```

Lastly we show that $tick_r$ (for any $r \in R^+$) preserves the claim. Since all that $tick_r$ can do is to advance *now* by r, it clearly preserves the claim. □

4. One more example: Fischer's protocol

We use one more example to demonstrate the proposed method. The example used is Fischer's protocol.

4.1. Fischer's protocol

Fischer's protocol is a real-time mutual exclusion algorithm among multiple processes, say $N\,(\geq 1)$ processes. The code executed by a process $i\,(1 \leq i \leq N)$ in a traditional style is as follows:

a:	Remainder Section	$[0, \infty]$
	loop	
b:	**repeat until** *turn* = 0	$[0, \infty]$
c:	*turn* := i	$[0, d1]$
d:	**if** *turn* = i **then break**	$[d2, \infty]$
cs:	Critical Section	$[0, \infty]$
e:	*turn* := 0	$[0, \infty]$

Initially *turn* is set to 0, and all processes are at location a. $[0, d1]$ forces process i to execute *turn* := i between 0 and *d1* time units after process i gets to location c, and any other pair specifies a similar timing constraint. Besides, we have the assumption that $0 < d1 < d2 < \infty$.

We model Fischer's protocol as a TBCM. In this modeling, we use six sets of transition rules: $\{try_i\}$, $\{test_i\}$, $\{set_i\}$, $\{check_i\}$, $\{exit_i\}$, and $\{reset_i\}$, where $1 \leq i \leq N$. try_i (or $exit_i$) means that process i begins executing the algorithm (or leaves Critical Section). The other transition rules $test_i$, set_i, $check_i$, and $reset_i$ stand for the statements at locations b, c, d, and e, respectively. The discrete variables used in this modeling are *turn* and $\{loc_i \mid 1 \leq i \leq N\}$. Discrete variable *turn* corresponds to *turn* in the algorithm, and loc_i shows the location at which process i is. The condition on which try_i ($test_i$, set_i, $check_i$, $exit_i$, or $reset_i$) gets effective is that loc_i is location a (b, c, d, cs, or e).

Since the minimum and maximum delays of try_i, $test_i$, $exit_i$, and $reset_i$ are 0 and ∞, respectively, their lower and upper bounds are not used. Besides, the minimum delay of set_i is 0, and the maximum delay of $check_i$ is ∞. Thus, l_{set_i} and u_{check_i} are not used either. As shown in the algorithm, the maximum delay of set_i is *d1* and the minimum delay of $check_i$ is *d2*. Hence, we use two sets of clock variables $\{u_{set_i}\}$ and $\{l_{check_i}\}$, where $1 \leq i \leq N$.

We also use one more clock variable *now* and a set of transition rules $\{tick_r \mid r \in R^+\}$.

4.2. Specification of Fischer's protocol in CafeOBJ

The main part of the signature is as follows:

```
*[Sys]*
-- any initial state
op   init             :           -> Sys
-- observations
bop  turn             : Sys       -> Nat
bop  loc              : Sys NzNat -> Loc
bop  now              : Sys       -> Real+
bop  l                : Sys NzNat -> Real+
bop  u                : Sys NzNat -> Timeval
-- actions
bops try test set     : Sys NzNat -> Sys
bops check exit reset : Sys NzNat -> Sys
bop  tick             : Sys Real+ -> Sys
-- delays
ops  d1 d2            :           -> Real+
```

Hidden sort `Sys` represents the state space of the TBCM, and visible sort `Loc` represents locations such as a and b. `u` and `l` correspond to u_{set_i} and l_{check_i}, respectively.

We have eight sets of equations in the specification: one for any initial state, and the others for the seven actions. In this paper, three sets of equations are presented: one for any initial state, and the others for `set` and `check`. In the specification, `S`, `I`, and `J` are CafeOBJ variables whose sorts are `Sys`, `NzNat`, and `NzNat`, respectively.

The equations defining any initial state are as follows:

```
eq turn(init)  = 0 .
eq loc(init,I) = a .
eq now(init)   = 0 .
eq l(init,I)   = 0 .
eq u(init,I)   = oo .
```

Since set_i and $check_i$ for any process $i \in \{1, \ldots, N\}$ are not effective in any initial state, u_{set_i} and l_{check_i} for any $i \in \{1, \ldots, N\}$ are initially set to 0 and ∞, respectively.

The equations defining how a state of the TBCM changes if set_i for any process $i \in \{1, \ldots, N\}$ is executed in the state are as follows:

```
ceq turn(set(S,I))  = I if loc(S,I) == c .
ceq turn(set(S,I))  = turn(S) if loc(S,I) =/= c .
ceq loc(set(S,I),I) = d if loc(S,I) == c .
ceq loc(set(S,I),J) = loc(S,J)
    if I =/= J or loc(S,I) =/= c .
eq  now(set(S,I))   = now(S) .
ceq l(set(S,I),I)   = now(S) + d2
    if loc(S,I) == c .
ceq l(set(S,I),J)   = l(S,J)
    if I =/= J or loc(S,I) =/= c .
ceq u(set(S,I),I)   = oo if loc(S,I) == c .
ceq u(set(S,I),J)   = u(S,J)
    if I =/= J or loc(S,I) =/= c .
```

If set_i is executed in a state in which it is effective, *turn* and loc_i are set to i and location d, respectively, and the other discrete variables, namely $loc_j\,(j \neq i)$, are left unchanged. Besides, u_{set_i} and l_{check_i} are set to ∞ and $now + d2$, respectively, because set_i and $check_i$ are effective and ineffective in the supposed state, and get ineffective and effective in the successor state, respectively. The other clock variables are left unchanged.

The equations defining how a state of the TBCM changes if $check_i$ for any process $i \in \{1, \ldots, N\}$ is executed in the state are as follows:

```
eq  turn(check(S,I))  = turn(S) .
ceq loc(check(S,I),I) = cs
    if loc(S,I) == d and turn(S) == I
       and l(S,I) <= now(S) .
ceq loc(check(S,I),I) = b
    if loc(S,I) == d and turn(S) =/= I
       and l(S,I) <= now(S) .
ceq loc(check(S,I),J) = loc(S,J)
    if I =/= J or loc(S,I) =/= d
       or now(S) < l(S,I) .
eq  now(check(S,I))   = now(S) .
```

```
ceq l(check(S,I),I)   = 0
    if loc(S,I) == d and l(S,I) <= now(S) .
ceq l(check(S,I),J)   = l(S,J)
    if I =/= J or loc(S,I) =/= d
       or now(S) < l(S,I) .
eq  u(check(S,I),J)   = u(S,J) .
```

If $check_i$ is executed in a state in which it is effective and $l_{check_i} \leq now$, loc_i is set to location cs or b depending on whether *turn* is i or not. In both cases, l_{check_i} is set to 0 because $check_i$ gets ineffective after the execution. The other variables are left unchanged.

4.3. Verification of Fischer's protocol with CafeOBJ

We show that Fischer's protocol satisfies mutual exclusion, namely that more than one process never enters its critical section simultaneously. Before the main claim is shown, two sub-claims are shown.

Claim 2. *In any reachable state, if* $loc_i = \mathrm{d}$, $turn = i$, *and* $loc_j = \mathrm{c}$, *then* $u_{set_j} < l_{check_i}$ *for any* $i, j \in \{1, \ldots, N\}$.

Proof. The claim is vacuously true in any initial state. Then, the claim is shown to be preserved by every transition rule. We prove this in the same way as the previous proof. In this paper, we present two cases which prove that set_i and $check_i$ for any $i \in \{1, \ldots, N\}$ preserve the claim.

First we show that set_i for any i preserves the claim. It is sufficient to consider a state in which loc_i is location c. However, if there is no process at location c in s', the claim is vacuously true in s'. Thus, suppose that there exists a process j ($\neq i$) at location c in s'. Since execution of set_i changes loc_i to location d and does not change loc_m for any other m, we also suppose that $loc_j = \mathrm{c}$ in s. All we have to do is to show that $u_{set_j} < l_{check_i}$ in s' because $turn = i$ in s'. The following proof score can be used to show it:

```
open FISCHER
  ops s s' : -> Sys .
  ops i j  : -> NzNat .
  eq  loc(s,i)                  = c .
  eq  loc(s,j)                  = c .
  eq  s'                        = set(s,i) .
  -- from Lemma 1
  eq  u(s,j) <= now(s) + d1     = true .
  -- from d1 < d2
  eq  now(s) + d1 < now(s) + d2 = true .
  vars X Y Z : Real+ .
  ceq [r1] : X < Z = true if X <= Y and Y < Z .
  start u(s',j) < l(s',i) .
  apply reduce at term .
  apply +.r1 with Y = now(s) + d1 at term .
  apply reduce at term .
close
```

In the proof score, we have two inequalities from Lemma 1 ($u_{set_j} \leq now + d1$ because set_j is effective in s) and the assumption that $d1 < d2$. In order to reduce the term `u(s',j) < l(s',i)`, we have to use the transitive rule of inequalities on real numbers. The transitive rule is given by the conditional equation labeled by `r1`. However, the conditional equation cannot be used as a rewrite rule as it is because the condition has CafeOBJ variable `Y` that does not appear in the left-hand side. Thus, we have to use CafeOBJ command `apply` to use the conditional equation. The second use of `apply` instructs the CafeOBJ system to use the conditional equation to rewrite the given expression.

Next we show that $check_i$ for any i preserves the claim. It is sufficient to consider a state in which $loc_i = \mathrm{d}$ and $l_{check_i} \leq now$. The case is divided into two sub-cases: 1) $turn = i$, and 2) $turn \neq i$.

In case (1), it is sufficient to show that process i is at location cs and *turn* is i in s'. The following proof score can be used to show that this is true:

```
open FISCHER
  ops s s' : -> Sys .
  op  i    : -> NzNat .
  eq  loc(s,i)          = d .    -- assumption.
  eq  turn(s)           = i .    -- assumption.
  eq  l(s,i) <= now(s) = true . -- assumption.
  eq  s'                = check(s,i) .
  red loc(s',i) == cs and turn(s') == i .
close
```

In case (2), it suffices that we only consider the case in which there exist a process j such that $loc_j = \mathrm{d}$ and $turn = j$ and a process k such that $loc_k = \mathrm{c}$ in s' because the claim is vacuously true in s' for any other case. If so, there must be such processes j and k in s as well. All we have to do is to show that l_{check_j} and u_{set_k} are left unchanged. The following proof score can be used to show that this is true:

```
open FISCHER
  ops s s'  : -> Sys .
  ops i j k : -> NzNat .
  eq  loc(s,i)          = d .    -- assumption.
  eq  loc(s,j)          = d .    -- assumption.
  eq  loc(s,k)          = c .    -- assumption.
  eq  turn(s)           = j .    -- assumption.
  eq  l(s,i) <= now(s) = true . -- assumption.
  eq  s'                = check(s,i) .
  red l(s',j) == l(s,j) and u(s',k) == u(s,k)
      and loc(s',i) == b .
close
```

The third conjunct of the term following `red` sees to it that process i moves to location b after the execution. □

Claim 3. *In any reachable state, if* $loc_i \in \{\mathrm{cs}, \mathrm{e}\}$, *then* $turn = i$ *and* $loc_j \neq \mathrm{c}$ *for any* $i, j \in \{1, \ldots, N\}$.

Proof. The claim is initially true. Then, the claim is shown to be preserved by every transition rule. We prove it in the same way as the previous proofs. In this paper, we present the two cases for set_i and $check_i$ for any $i \in \{1, \ldots, N\}$.

First we show that set_i for any i preserves the claim. It is sufficient to consider a state in which $loc_i = \mathrm{c}$. The case is divided into two sub-cases: 1) there exists a process k at location cs or e, and 2) there exists no such a process.

In case (1), there exists no process at location c from the hypothesis, which contradicts the assumption that $loc_i = \mathrm{c}$.

In case (2), since execution of set_i does not change loc_j for any process j except i, all we have to do is to show that

loc_i is set to location d as set_i is executed, which means that there is still no process at location cs or e in s'. The following proof score can be used to show it:

```
open FISCHER
  ops s s' : -> Sys .
  op  i    : -> NzNat .
  eq  loc(s,i) = c . -- assumption.
  eq  s'       = set(s,i) .
  red loc(s',i) == d .
close
```

Next we show that $check_i$ for any i preserves the claim. It is sufficient to consider a state in which $loc_i = \mathrm{d}$ and $l_{check_i} \leq now$. The case is divided into two sub-cases (1) and (2) as well.

In case (1), $turn = k\,(\neq i)$ from the hypothesis. Since $check_i$ does not change *turn* and loc_j for any process j except i, all that is needed is to show that loc_i is set to location b. The following proof score can be used to show it:

```
open FISCHER
  ops s s' : -> Sys .
  op  i    : -> NzNat .
  eq  loc(s,i)         = d .    -- assumption.
--  eq  turn(s) =/= i .         -- assumption.
  eq  l(s,i) <= now(s) = true . -- assumption.
  eq  s'               = check(s,i) .
  red loc(s',i) == b .
close
```

If we have no equation for *turn* in s, *turn* is implicitly treated as not being equal to i in s. The comment in the proof score is used to remind us that $turn \neq i$.

In case (2), if $turn \neq i$, then loc_i is set to location b as shown in the previous proof score, which means that the claim still holds in s'. Thus, suppose that $turn = i$. Besides, if we suppose that there exists a process j such that $loc_j = c$, then $now < l_{check_i}$ from Lemma 1 ($now \leq u_{set_j}$) and Claim 2 ($u_{set_j} < l_{check_i}$), which contradicts the assumption that $l_{check_i} \leq now$. Hence, suppose that there exists no process j such that $loc_j = c$. Since $check_i$ does not change *turn* and loc_j for any other j, all we have to do is to show that loc_i is set to location cs as $check_i$ is executed. The following proof score can be used to show it:

```
open FISCHER
  ops s s' : -> Sys .
  op  i    : -> NzNat .
  eq  loc(s,i)         = d .    -- assumption.
  eq  turn(s)          = i .    -- assumption.
  eq  l(s,i) <= now(s) = true . -- assumption.
  eq  s'               = check(s,i) .
  red loc(s',i) == cs and turn(s') == i .
close
```

The second conjunct of the term following `red` sees to it that $turn = i$ as well after the execution. □

Claim 4 (mutual exclusion). *In any reachable state, if* $loc_i = \mathrm{cs}$ *and* $loc_j = \mathrm{cs}$, *then* $i = j$ *for any* $i, j \in \{1, \ldots, N\}$.

Proof. The claim immediately follows from Claim 3. □

5 Concluding remarks

The definition of a TBCM is mainly affected by specification and verification of distributed real-time systems with TLA [1] and with I/O automata [8], and also by clocked transition systems [7]. How to describe a TBCM in CafeOBJ is similar to general timed automata described in [8].

The asynchronous data sending problem is a simplified version of the lossy-queue example presented in [1]. Fischer's protocol was designed by M. Fischer. It has been used to demonstrate that formal methods can reason about timing-based systems. Proofs that it has mutual exclusion are presented in [1, 7, 8]. They are similar to our proof.

The two examples given in this paper are probably the first attempt to specify distributed real-time systems in CafeOBJ and verify their (safety) properties with the help of the CafeOBJ system.

The proofs presented in this paper comprise CafeOBJ proof scores and narrative explanations. That is, formalities and informalities are mingled together. Such proof styles may be called semi-formal, which has been advocated by Goguen and is adopted in the UCSD Tatami project[6, 13]. But, distributed real-time systems are out of the scope of the project.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM TOPLAS*, 16(5):1543–1571, 1994.

[2] CafeOBJ web page. http://www.ldl.jaist.ac.jp/cafeobj/.

[3] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley, Reading, MA, 1988.

[4] R. Diaconescu and K. Futatsugi. *CafeOBJ report*. AMAST Series in Computing, 6. World Scientific, Singapore, 1998.

[5] J. Goguen and G. Malcolm. A hidden agenda. *Theoret. Comput. Sci.*, 245:55–101, 2000.

[6] J. A. Goguen and K. Lin. Web-based support for cooperative software engineering. *Annals of Software Engineering*, 12, 2001 (to apper).

[7] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. *Acta Informatica*, 36:837–912, 2000.

[8] N. A. Lynch. *Distributed algorithms*. Morgan-Kaufmann, San Francisco, CA, 1996.

[9] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, NY, 1995.

[10] K. Ogata and K. Futatsugi. Formal verification of the MCS list-based queuing lock. In *Proc. of ASIAN '99*, LNCS 1742, pages 281–293. Springer-Verlag, 1999.

[11] K. Ogata and K. Futatsugi. Formally modeling and verifying Ricart&Agrawala distributed mutual exclusion algorithm. In *Proc. of APAQS '01*. IEEE CS Press, 2001 (to appear).

[12] T. Seino, K. Ogata, and K. Futatsugi. Specification and verification of a single-track railroad signaling in CafeOBJ. *IEICE Trans. Fundamentals*, E84-A(6):1471–1478, 2001.

[13] The UCSD Tatami project web page. http://www-cse.ucsd.edu/groups/tatami/.