Abstract

CafeOBJ is a wide spectrum specification language based on multiple logical foundations. CafeOBJ can be used to specify dynamic as well as static aspects of systems including object-oriented and reactive systems and verify their properties with the help of the CafeOBJ system. In this paper, we show that CafeOBJ can be also used to describe real-time systems and verify their properties. Concretely, we evolve UNITY computational models by introducing so-called clock variables so as to model real-time systems, describe a specification of a railroad crossing system in CafeOBJ and verify that the system has a safety property based on the specification with the help of the CafeOBJ system.

1. Introduction

CafeOBJ [3, 5] is a wide spectrum specification language based on multiple logical foundations. It is one of its most distinctive features that it is possible to naturally describe objects in object-orientation in terms of their behavior and verify their properties as well as to describe abstract data types such as integers and verify their properties in CafeOBJ alone. Besides, since objects can be regarded as transition systems such as UNITY [4] computational models, CafeOBJ can be also used to describe reactive systems and verify their properties. Actually we have shown before that CafeOBJ can be well used to describe reactive systems and verify their properties by giving some examples [14, 13]. In [14, 13], UNITY computational models are basically used to model reactive systems.

In this paper, we show that CafeOBJ can be also used to describe real-time systems and verify their properties. Concretely, we describe a specification of a railroad crossing system in CafeOBJ and verify that the system has a safety property based on the specification with the help of the CafeOBJ system. In the description of the railroad crossing system, we use UNITY computational models that is evolved so as to model real-time systems because we have known that UNITY computational models are suited for specification of reactive systems in CafeOBJ from our experience.

The rest of the paper is organized as follows. In Sect. 2, we first reformulate UNITY computational models in the same manner as the definition of fair transition systems [11, 12]. The reformulated computational models are called basic computational models, or BCMS. We then evolve BCMS into timed extension of basic computational models, or TBCMS by introducing so-called clock variables so as to model real-time systems. Section 3 describes how to describe TBCM modeling real-time systems in CafeOBJ. In Sect. 4, we first describe what the railroad crossing system is and how to model the system as a TBCM. We next describe how to specify the TBCM modeling the system in CafeOBJ, and then verify that the system has a safety property based on the specification with the help of the CafeOBJ system. Section 5 concludes the paper.

2. Timed extension of UNITY computational models

2.1. Basic computational models: transition systems

UNITY [4] computational models, or transition systems are basically used as basic computational models (BCMS) in order to model reactive systems. A BCM $S = \langle V, I, T \rangle$ consists of:

- $V$: A finite set of variables. Each variable has its own type. The variables (or their possible values) form the state space $\Sigma$ of $S$, and a state of $S$ is a point, or an element of $\Sigma$.
- $I$: The initial condition. This condition specifies the initial values of the variables. Since some variables may not be specified by $I$, $S$ may have more than one initial state.
\( \mathcal{T} \): A set of transition rules. Each transition rule \( \tau \in \mathcal{T} \) is a function \( \tau : \Sigma \rightarrow \Sigma \) mapping each state \( s \in \Sigma \) into a successor state \( \tau(s) \in \Sigma \).

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 sequence 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 infinitely exist indexes \( \tilde{i} \in \{0, 1, \ldots \} \) such that \( s_{\tilde{i}+1} = \tau(s_{\tilde{i}}) \).

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

Transition rules are generally defined together with conditions on which the transition rules are effectively executed, namely that their execution can change states of BCMS. The concept effectiveness is similar to enabledness used in description of transition systems in temporal logic such as TLA [9] or in a precondition-effect style such as I/O automata [10]. If the condition of a transition rule is false in a state of a BCM, namely that the transition rule is not effective in the state, its execution does not change the state.

### 2.2. Timed extension of basic computational models

As many approaches to modeling real-time systems as transition systems [1, 2, 7, 8, 10], BCMS are evolved so as to model real-time systems by introducing so-called clock variables. The evolved version of BCMS is called timed extension of basic computational models (TBCMs). A TBCM \( \mathcal{S} = (\mathcal{V}, \mathcal{I}, \mathcal{T}) \) consists of:

- \( \mathcal{V} \): A finite 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 \( (\mathbb{R}^+) \) or infinity (\( \infty \)). For each \( \tau \in \mathcal{T} \) except \( \text{tick} \) (described later), there are two clock variables \( l_\tau : \mathbb{R}^+ \) and \( u_\tau : (\mathbb{R}^+ \setminus \{0\}) \cup \{\infty\} \). They are basically used to force \( \tau \) to be executed between lower bound \( l_\tau \) and upper bound \( u_\tau \). Besides, there is one special clock \( \text{now} : \mathbb{R}^+ \). It serves as the master clock and always indicates the elapsed time after a system starts its computation.
- \( \mathcal{I} \): The initial condition as a BCM. Master clock now is initially set to 0.
- \( \mathcal{T} \): A set of transition rules as a BCM. But, there is time advancing transition rule \( \text{tick} : \mathcal{V} \rightarrow \mathcal{V} \). It advances now some amount unless now exceeds upper bound \( u_\tau \) for any \( \tau \in \mathcal{T} \) but \( \text{tick} \). \( \text{tick} \) does not change any variable but \( \text{now} \), and any transition rule but \( \text{tick} \) does not affect advancing \( \text{now} \). For each \( \tau \in \mathcal{T} \) but \( \text{tick} \), in addition to the condition on which \( \tau \) gets effective, \( l_\tau \leq \text{now} \) is given to \( \tau \) as a conjunct of its condition.

For each \( \tau \in \mathcal{T} \) except \( \text{tick} \), besides two clock variables \( l_\tau \) and \( u_\tau \), there are two constants \( d_{\min}, \) and \( d_{\max} \) whose types are the same as \( l_\tau \) and \( u_\tau \), respectively. \( d_{\min} \) and \( d_{\max} \) are called the minimum and maximum delays of \( \tau \). If \( \tau \) is initially effective, \( l_\tau \) and \( u_\tau \) are initially set to \( d_{\min} \) and \( d_{\max} \), respectively, and otherwise \( l_\tau \) and \( u_\tau \) are initially set to 0 and \( \infty \), respectively. Suppose that \( \tau \) is executed in a state \( s \) in which it is effective and \( l_\tau \leq \text{now} \), and let \( \text{now}' \) be the successor state. If \( \tau \) keeps effective in \( s' \), \( l_\tau \) and \( u_\tau \) are set to \( \text{now} + d_{\min} \) and \( \text{now} + d_{\max} \), respectively, and otherwise \( l_\tau \) and \( u_\tau \) are set to 0 and \( \infty \), respectively. For any other transition rule \( \tau' \), if it is non-effective (or effective) in \( s \) and it gets effective (or non-effective) in \( s' \), then \( l_{\tau'} \) and \( u_{\tau'} \) are set to \( \text{now} + d_{\min} \) (or 0) and \( \text{now} + d_{\max} \) (or \( \infty \)), respectively. Otherwise, \( l_{\tau'} \) and \( u_{\tau'} \) are left unchanged.

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

- **Time Divergence**: As \( i \) increases, \( \text{now} \) 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 that a TBCM \( \mathcal{S} \) is non-Zeno is that for each \( \tau \in \mathcal{T} \) except \( \text{tick} \), \( l_\tau \) is always less than or equal to \( u_\tau \), i.e., \( d_{\min} \leq d_{\max} \) (from [1]). If for some \( \tau \in \mathcal{T} \), \( d_{\min} \) (or \( d_{\max} \)) is 0 (or \( \infty \)), then \( l_\tau \) (or \( u_\tau \)) may be removed from \( \mathcal{V} \).

In the verification given in the remainder of the paper, we use the following lemma (from [10]):

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

- \( \text{now} \leq u_{\tau} \) for each \( \tau \in \mathcal{T} \) but \( \text{tick} \).

### 3. Description of TBCMs in CafeOBJ

We use TBCMs as models of real-time systems to specify such systems in CafeOBJ.

CafeOBJ is mainly based on two logical foundations: initial and hidden algebra. Initial algebra is used to specify
abstract data types such as integers, and hidden algebra [6] to specify objects in object-orientation. There are two kinds of sorts (corresponding to types in programming languages) in CafeOBJ. They are visible and hidden sorts. A visible sort represents an abstract data type, and a hidden sort a set of states of an object. There are basically two kinds of operations to hidden sorts. They are action and observation operations, corresponding to so-called methods in object-orientation. An action operation, or an action can change a state of an object. It takes a state of an object and zero or more data, and returns another (possibly the same) state of the object. An observation operation, or an observation can be used to observe the value of a data component in an object. It takes a state of an object and zero or more data, and returns the value of a data component in the object. Both actions and observations are defined with equations.

Since objects can be regarded as transition systems, TBCMs can be naturally described in CafeOBJ. Discrete and clock variables are represented by observations, and their initial values are specified with equations. Transition rules are represented by actions, and their behavior is also specified with equations.

In specification of a TBCM in CafeOBJ, we first write the signature of the specification of the TBCM, declaring sorts and operations, next write equations defining the initial values of the observations, and then write equations defining how a state of the TBCM changes after each action is executed in that state.

4. Railroad crossing

Let us consider the following scenario. A train is running to a railroad crossing and it signals to a controller that it is approaching to the railroad crossing at least \( d1 \) time units before it enters the railroad crossing. As \( d2 \) time units is elapsed after the signal, the controller instructs the gate to close. The gate gets closed within \( d3 \) time units. After the train goes out the railroad crossing, it signals to the controller that it has left the railroad crossing. After the controller receives the signal, it instructs the gate to open. Then the gate gets opened. We show that if \( 0 < d2 + d3 < d1 < \infty \), then the gate is closed whenever the train is passing over the railroad crossing.

4.1. Modeling

We model the scenario, or the railroad crossing system as a TBCM. In this modeling, there are three discrete variables \( t, g, \) and \( c \) that show the states of the train, the gate, and the controller, respectively.

The states of the train are sect0, sect1, crossing, sect2, and sect3. If the train is running to the railroad crossing but far away from the railroad crossing, its state is sect0. If the train is approaching near enough to the railroad crossing to signal to the controller about that, its state is sect1. If the train is passing over the railroad crossing, its state is crossing. Just after the train leaves the railroad crossing, its state is sect2 until the train signals to the controller that it has left the railroad crossing. After the signal, its state is sect3.

The states of the gate are up, lowering, down, and raising. If the gate is open (or closed), its state is up (or down). If the gate is getting closed (or opened), its state is lowering (or raising).

The states of the controller are state0, state1, state2, and state3. If the controller waits for the train to signal that the train is approaching to the railroad crossing, its state is state0. After the signal, its state is state1 until the controller instructs the gate to close. After the instruction, its state is state2 until the train signals to the controller that the train has left the railroad crossing. After the signal, its state is state3 until the controller instructs the gate to open. After the instruction, the controller gets state0 and waits for the next train to signal that the train is approaching to the railroad crossing.

Except for tick, there are eight transition rules. The following show the eight transition rules. The following show the eight transition rules, 1) what each transition rule corresponds to in the scenario, 2) the condition on which each transition rule gets effective, 3) the minimum and maximum delays for each transition rule, and 4) the states of the train, the gate, and/or the controller (if they change) after each transition rule is executed if the transition rule is effective and its timing constraint is satisfied.

- **approach**: 1) The train signals to the controller that it is approaching to the railroad crossing. 2) The states of the train and the controller are sect0 and state0, respectively. 3) The minimum and maximum delays are 0 and \( \infty \), respectively. 4) The states of the train and the controller get sect1 and state1, respectively.
- **in**: 1) The train enters the railroad crossing. 2) The state of the train is sect1. 3) The minimum and maximum delays are \( d1 \) and \( \infty \), respectively. 4) The state of the train gets crossing.
- **out**: 1) The train leaves the railroad crossing. 2) The state of the train is crossing. 3) The minimum and maximum delays are 0 and \( \infty \), respectively. 4) The state of the train gets sect2.
- **exit**: 1) The train signals to the controller that the train has left the railroad crossing. 2) The states of the train and the controller are sect2 and state2. 3) The minimum and maximum delays are 0 and \( \infty \), respectively. 4) The states of the train and the controller get sect3 and state3, respectively.
down: 1) The gate gets closed. 2) The state of the gate is lowering. 3) The minimum and maximum delays are 0 and d3. 4) The state of the gate gets down.

up: 1) The gate gets opened. 2) The state of the gate is raising. 3) The minimum and maximum delays are 0 and \( \infty \), respectively. 4) The state of the gate gets up.

lower: 1) The controller instructs the gate to close. 2) The states of the gate and the controller are up and state1, respectively. 3) The minimum and maximum delays are \( d2 \) and \( d2 \), respectively. 4) The states of the gate and the controller get lowering and state2, respectively.

raise: 1) The controller instructs the gate to open. 2) The states of the gate and the controller are down and state3, respectively. 3) The minimum and maximum delays are 0 and \( \infty \), respectively. 4) The states of the gate and the controller get raising and state0, respectively.

Except for now, we use four clock variables. As it is clear from the above descriptions of the eight transition rules, they are \( l_{in} \), \( u_{down} \), \( l_{lower} \), and \( u_{lower} \).

Initially, the states of the train, the gate, and the controller are \( state0 \), \( up \), and \( state0 \), respectively.

4.2. Specification

We describe how to specify the TBCM modeling the railroad crossing system in CafeOBJ.

The main part of the signature is as follows:

```plaintext
-- any initial state
op init : -> Sys
-- observations
bop t : Sys -> TState
bop g : Sys -> GState
bop c : Sys -> CState
bop now : Sys -> Real+
bop cl : Sys -> Real+
bop gu cu : Sys -> Timeval
-- actions
bops approach in out exit : Sys -> Sys
bops down up lower raise : Sys -> Sys
bop tick : Sys Real+ -> Sys
-- delays
ops d1 d2 d3 : -> Real+
```

A comment starts with ‘--’ and terminates at the end of the line. Hidden sort Sys represents the state space of the TBCM, and visible sorts TState, GState, CState, Real+, and Timeval represent the states of the train, the states of the gate, the states of the controller, non-negative real numbers (\( \mathbb{R}^+ \)), and \( (\mathbb{R}^+ \setminus \{0\}) \cup \{\infty\} \), respectively. Operations with no arguments are called constants. Constant init denotes any initial state of the TBCM. Clock variables \( l_{in} \), \( u_{down} \), \( l_{lower} \), and \( u_{lower} \) correspond to observations \( t_1 \), \( gu \), \( cl \), and \( cu \), respectively. Except for these observations, it is clear which transition rules, variables, and delays correspond to which actions, observations, and constants, respectively.

We have basically 10 sets of equations in the specification: one for any initial state, and the others for the nine actions. The equations defining any initial state are as follows:

```plaintext
eq t(init) = sect0.
eq g(init) = up.
eq c(init) = state0.
eq now(init) = 0.
eq tl(init) = 0.
eq gu(init) = oo.
eq cl(init) = 0.
eq cu(init) = oo.
```

Constant \( \infty \) stands for \( \infty \). Since transition rules \( in \), \( down \), and \( lower \) are not effective in any initial state, \( l_{in} \), \( u_{down} \), \( l_{lower} \), and \( u_{lower} \) are set to 0, \( \infty \), 0, and \( \infty \), respectively.

The equations defining how a state of the TBCM changes if transition rule approach is executed in the state are as follows:

```plaintext
ceq t(approach(S)) = sect1.
if t(S) == sect0 and c(S) == state0.
ceq t(approach(S)) = t(S).
if t(S) == sect0 or c(S) == state0.
eq g(approach(S)) = g(S).
eq c(approach(S)) = state1.
if t(S) == sect0 and c(S) == state0.
eq c(approach(S)) = state0.
eq cl(approach(S)) = cl(S).
eq cl(approach(S)) = now(S) + d1.
if t(S) == sect0 and c(S) == state0.
eq cu(approach(S)) = gu(S).
eq cu(approach(S)) = now(S) + d2.
if t(S) == sect0 and c(S) == state0.
eq cl(approach(S)) = cl(S).
eq cl(approach(S)) = now(S) + d2.
if t(S) == sect0 and c(S) == state0.
```

If transition rule approach is executed in a state in which it is effective, transition rules \( in \) and \( lower \) newly get effective. Thus, if so, \( l_{in} \) (or \( t_1 \)), \( l_{lower} \) (or \( cl \)), and \( u_{lower} \) (or \( cu \)) are set to \( now + d1 \), \( now + d2 \), and \( now + d2 \), respectively.

The equations defining how a state of the TBCM changes if transition rule \( in \) is executed in the state are as follows:

```plaintext
ceq t(in(S)) = crossing.
if t(S) == sect1 and tl(S) <= now(S).
ceq t(in(S)) = t(S).
if t(S) == sect1 or now(S) < tl(S).
eq g(in(S)) = g(S).
eq c(in(S)) = c(S).
eq now(in(S)) = now(S).
ceq cl(in(S)) = cl(S).
if t(S) == sect1 and tl(S) <= now(S).
ceq cl(in(S)) = tl(S).
if t(S) == sect1 or now(S) < tl(S).
eq gu(in(S)) = gu(S).
eq cl(in(S)) = cl(S).
eq cu(in(S)) = cu(S).
```
If transition rule in is executed in a state in which it is effective and $l_{in} \leq now$, transition rule in gets non-effective again. Thus, if so, $l_{in}$ (or $t_{1}$) is set to 0.

The equations defining how a state of the TBCM changes if transition rule out is executed in the state are as follows:

- $ceq t(out(S)) = sect2$ if $t(S) = crossing$.
- $ceq t(out(S)) = t(S)$ if $t(S) /= crossing$.
- $eq g(out(S)) = g(S)$.
- $eq c(out(S)) = c(S)$.
- $eq now(out(S)) = now(S)$.
- $eq t(\ell(out(S))) = t(\ell(S))$.
- $eq gu(out(S)) = gu(S)$.
- $eq cl(out(S)) = cl(S)$.
- $eq cu(out(S)) = cu(S)$.

The equations defining how a state of the TBCM changes if transition rule exit is executed in the state are as follows:

- $ceq t(\ell(\ell(S))) = sect3$ if $t(S) = sect2$ and $c(S) = state2$.
- $ceq t(\ell(\ell(S))) = t(S)$ if $t(S) =/= sect2$ or $c(S) =/= state2$.
- $eq g(\ell(\ell(S))) = g(S)$.
- $eq c(\ell(\ell(S))) = state3$ if $t(S) = sect2$ and $c(S) = state2$.
- $ceq c(\ell(\ell(S))) = c(S)$ if $t(S) =/= sect2$ or $c(S) =/= state2$.
- $eq now(\ell(\ell(S))) = now(S)$.
- $eq t(\ell(\ell(S))) = t(\ell(S))$.
- $eq gu(\ell(\ell(S))) = gu(S)$.
- $eq cl(\ell(\ell(S))) = cl(S)$.
- $eq cu(\ell(\ell(S))) = cu(S)$.

The equations defining how a state of the TBCM changes if transition rule down is executed in the state are as follows:

- $eq t(down(S)) = t(S)$.
- $ceq g(down(S)) = down$ if $g(S) = lowering$.
- $ceq c(down(S)) = c(S)$ if $g(S) =/= lowering$.
- $eq now(down(S)) = now(S)$.
- $eq t(\ell(down(S))) = t(\ell(S))$.
- $eq gu(down(S)) = gu(S)$ if $g(S) =/= lowering$.
- $eq cl(down(S)) = cl(S)$.
- $eq cu(down(S)) = cu(S)$.

If transition rule down is executed in a state in which it is effective, it newly gets non-effective. Thus, if so, $u_{down}$ (or gu) is set to $\infty$.

The equations defining how a state of the TBCM changes if transition rule up is executed in the state are as follows:

- $eq t(up(S)) = t(S)$.
- $ceq g(up(S)) = up$ if $g(S) = raising$.
- $ceq c(up(S)) = c(S)$ if $g(S) =/= raising$.
- $eq now(up(S)) = now(S)$.
- $eq t(\ell(up(S))) = t(\ell(S))$.
- $eq gu(up(S)) = gu(S)$.
- $eq cl(up(S)) = cl(S)$.
- $eq cu(up(S)) = cu(S)$.

The equations defining how a state of the TBCM changes if transition rule lower is executed in the state are as follows:

- $eq t(lower(S)) = t(S)$.
- $ceq g(lower(S)) = lowering$ if $g(S) = up$ and $c(S) = state1$ and $cl(S) = now(S)$.
- $ceq g(lower(S)) = g(S)$ if $g(S) =/= up$ or $c(S) =/= state1$.
- $eq now(lower(S)) = now(S)$.
- $eq t(\ell(lower(S))) = t(\ell(S))$.
- $eq gu(lower(S)) = gu(S)$ if $g(S) =/= up$ or $c(S) =/= state1$ and $cl(S) = now(S)$.
- $ceq cl(lower(S)) = 0$ if $g(S) = up$ and $c(S) = state1$ and $cl(S) = now(S)$.
- $ceq cl(lower(S)) = cl(S)$ if $g(S) =/= up$ or $c(S) =/= state1$ and $now(S) = cl(S)$.
- $ceq cu(lower(S)) = cu(S)$ if $g(S) =/= up$ or $c(S) =/= state1$ and $now(S) = cl(S)$.
- $eq gu(lower(S)) = gu(S)$ if $g(S) =/= up$ or $c(S) =/= state1$ and $now(S) = cl(S)$.
- $eq cl(lower(S)) = cl(S)$ if $g(S) =/= up$ or $c(S) =/= state1$ and $now(S) = cl(S)$.
- $eq cu(lower(S)) = cu(S)$.

If transition rule lower is executed in a state in which it is effective and $l_{lower} \leq now$, transition rules down and lower newly get effective and non-effective, respectively. Thus, if so, $u_{down}$ (or gu), $l_{lower}$ (or cl), and $u_{lower}$ (or cu) are set to $now + D$, 0, and $\infty$, respectively.

The equations defining how a state of the TBCM changes if transition rule raise is executed in the state are as follows:

- $eq t(raise(S)) = t(S)$.
- $ceq g(raise(S)) = raising$ if $g(S) =/= down$ and $c(S) = state3$.
- $ceq g(raise(S)) = g(S)$ if $g(S) =/= down$ or $c(S) =/= state3$.
- $ceq c(raise(S)) = state0$ if $g(S) =/= down$ and $c(S) = state3$.
- $ceq c(raise(S)) = c(S)$ if $g(S) =/= down$ or $c(S) =/= state3$.
- $eq now(raise(S)) = now(S)$.
- $eq t(\ell(raise(S))) = t(\ell(S))$.
- $eq gu(raise(S)) = gu(S)$.
- $eq cl(raise(S)) = cl(S)$.
- $eq cu(raise(S)) = cu(S)$.

The equations defining how a state of the TBCM changes if transition rule raise is executed in the state are as follows:

- $eq t(tick(S,D)) = t(S)$.
- $eq g(tick(S,D)) = g(S)$.
- $eq c(tick(S,D)) = c(S)$.
- $ceq now(tick(S,D)) = now(S) + D$ if now(S) + D <= gu(S) and now(S) + D <= cu(S).
- $ceq gu(tick(S,D)) = gu(S)$ if gu(S) < now(S) + D or cu(S) < now(S) + D.
- $eq t(\ell(tick(S,D))) = t(\ell(S))$.
- $eq gu(tick(S,D)) = gu(S)$.
- $eq cl(tick(S,D)) = cl(S)$.
- $eq cu(tick(S,D)) = cu(S)$.

Transition rule tick advances master clock now provided that now does not go beyond any upper bound, i.e. $u_{down}$ (or gu) and $u_{lower}$ (or cu).
4.3. Verification

We show that the railroad crossing system has the safety property that the gate is closed whenever the train is passing over the railroad crossing. Before the main claim is shown, seven sub-claims are shown.

**Claim 1.** In any reachable state, if \( t = \text{sect0} \) and \( c = \text{state0} \), then \( g = \text{up} \).

*Proof.* The claim is vacuously true in any initial state. Therefore, it suffices that the claim is shown to be preserved by every transition rule. Suppose that the claim holds in a state \( s \), we show that it holds keeping in the successor state \( s' \) after any transition rule is executed in \( s \). Note that the beginning of the proof is the same as in all the remaining proofs except Claims 6 and 8 and we omit it from them.

We show that transition rule \( \text{approach} \) preserves the claim. It is sufficient to consider a state in which \( \text{approach} \) is effective because even if it is executed in a state in which it is not effective, nothing changes. Thus, we consider a state in which \( t = \text{sect0} \), \( c = \text{state0} \), and the value of any other variable is arbitrary. The following proof score can be used to show that \( \text{approach} \) preserves the claim.

```
open CROSING
  ops s s' : -> Sys.
  eq t(s) = sect0.
  eq g(s) = up.
  eq c(s) = state0.
  eq s' = approach(s).
  red t(s') == sect1 and g(s') == up
    and c(s') == state1.
close
```

We show that transition rule \( \text{down} \) preserves the claim. It is sufficient to consider a state in which \( \text{down} \) is effective. Thus, we consider a state in which \( g = \text{lowering} \). Besides, from the hypothesis, \( t \neq \text{sect0} \) or \( c \neq \text{state0} \). The following proof score can be used to show that \( \text{down} \) preserves the claim.

```
open CROSING
  ops s s' : -> Sys.
  ~-- t(s) = sect0 or c(s) = state0.
  eq g(s) = lowering.
  eq s' = down(s).
  red t(s') == t(s) and g(s') == down
    and c(s') == c(s).
close
```

In CafeOBJ specifications, unless you explicitly define one thing as being equal to another thing in terms of equations, the two things are basically different. In the above proof score, we state that \( t \neq \text{sect0} \) or \( c \neq \text{state0} \) in state \( s \) without any equations about \( t \) and \( \text{sect0} \), or \( c \) and \( \text{state0} \).

We show that transition rule \( \text{tick} \) preserves the claim. Since \( \text{tick} \) only advances now, it clearly preserves the claim.

We can show that any other transition rule preserves the claim in the same way.

**Claim 2.** In any reachable state, if \( g = \text{raising} \), then \( c \neq \text{state1} \).

*Proof.* We show that transition rule \( \text{approach} \) preserves the claim. It is sufficient to consider a state in which \( \text{approach} \) is effective. Thus, we consider a state in which \( t = \text{sect0} \) and \( c = \text{state0} \). Besides, from Claim 1, \( g = \text{up} \) in this state as well. The following proof score can be used to show that \( \text{approach} \) preserves the claim.

```
open CROSING
  ops s s' : -> Sys.
  eq t(s) = sect0.
  eq g(s) = up.
  eq c(s) = state0.
  eq s' = approach(s).
  red t(s') == sect1 and g(s') == up
    and c(s') == state1.
close
```

We can show that any other transition rule preserves the claim in the same way.

**Claim 3.** In any reachable state, if \( g = \text{up} \) and \( c = \text{state1} \), then \( t = \text{sect1} \) and there exists a non-negative real number \( x \) such that \( l_{\text{lower}} = u_{\text{lower}} = x + d2 \) and \( l_{\text{in}} = x + d1 \).

*Proof.* We show that transition rule \( \text{in} \) preserves the claim. It is sufficient to consider a state in which \( \text{in} \) is effective and its timing constraint is satisfied. Thus, we consider a state in which \( t = \text{sect1} \) and \( l_{\text{in}} \leq \text{now} \). The case is divided into two sub-cases: 1) \( g = \text{up} \) and \( c = \text{state1} \), and 2) \( g \neq \text{up} \) or \( c \neq \text{state1} \).

In case (1), from the hypothesis, there exists a non-negative real number \( x \) such that \( u_{\text{lower}} = x + d2 \) and \( l_{\text{in}} = x + d1 \). Besides, from Lemma 1 (i.e. \( \text{now} \leq u_{\text{lower}} \)) and the assumption that \( d2 + d3 < d1 \), \( \text{now} < l_{\text{in}} \), which contradicts the assumption that \( l_{\text{in}} \leq \text{now} \).

In case (2), the following proof score can be used to show that \( \text{in} \) preserves the claim.

```
open CROSING
  ops s s' : -> Sys.
  eq t(s) = sect1.
  ~-- g(s) = up or c(s) = state1.
  eq t(l(s)) = now(s) = true.
  eq s' = in(s).
  red t(s') == crossing and g(s') == g(s)
    and c(s') == c(s).
close
```

We show that transition rule \( \text{up} \) preserves the claim. It is sufficient to consider a state in which \( \text{up} \) is effective. Thus, we consider a state in which \( g = \text{raising} \). Besides, from Claim 2, \( c \neq \text{state1} \). the following proof score can be used to show that \( \text{in} \) preserves the claim.

```
open CROSING
  ops s s' : -> Sys.
  ~-- c(s) = state1.
  eq g(s) = raising.
  eq s' = up(s).
  red g(s') == up and c(s') == c(s).
close
```

We can show that any other transition rule preserves the claim in the same way.
Claim 4. In any reachable state, if $g$ = lowering and $c$ = state2, then there exists a non-negative real number $x$ such that $u_{down} = x + d2 + d3$ and $l_{in} = x + d1$.

Proof. We show that transition rule in preserves the claim. It is sufficient to consider a state in which in is effective and its timing constraint is satisfied. Thus, we consider a state in which $t = sect1$ and $l_{in} \leq now$. The case is divided into two sub-cases: 1) $g$ = lowering and $c$ = state2, and 2) $g$ $\neq$ lowering or $c$ $\neq$ state2.

In case (1), from the hypothesis, there exists a non-negative real number $x$ such that $u_{down} = x + d2 + d3$ and $l_{in} = x + d1$. Besides, from Lemma 1 (i.e. now $\leq u_{down}$) and the assumption that $d2 + d3 < d1$, now $< l_{in}$, which contradicts the assumption that $l_{in} \leq now$.

In case (2), the following proof score can be used to show that in preserves the claim.

We show that transition rule lower preserves the claim. It is sufficient to consider a state in which lower is effective and its timing constraint is satisfied. Thus, we consider a state in which $g$ = up, $c$ = state1, and $l_{lower} \leq now$. From Claim 3, there exists a non-negative real number $x$ such that $l_{lower} = u_{lower} = x + d2$ and $l_{in} = x + d1$. Besides, from Lemma 1, now $\leq u_{lower}$. Hence, now $= x + d2$. The following proof score can be used to show that lower preserves the claim.

We can show that any other transition rule preserves the claim in the same way. □

Claim 5. In any reachable state, if $t = sect1$, then $g$ = up and $c$ = state1, $g$ = lowering and $c$ = state2, or $g$ = down and $c$ = state2.

Proof. We show that transition rule down preserves the claim. It is sufficient to consider a state in which down is effective. Thus, we consider a state in which $g$ = lowering.

The case is divided into two sub-cases: 1) $t = sect1$, and 2) $t \neq sect1$.

In case (1), from the hypothesis, $c$ = state2 as well. The following proof score can be used to show that down preserves the claim in this case.

We show that transition rule lower preserves the claim. It is sufficient to consider a state in which lower is effective and its timing constraint is satisfied. Thus, we consider a state in which $g$ = up, $c$ = state1, and $l_{lower} \leq now$. From Claim 3, $t = sect1$ in this state as well. The following proof score can be used to show that lower preserves the claim.

We can show that any other transition rule preserves the claim in the same way. □

Claim 6. In any reachable state, if transition rule in is effective and its timing constraint is satisfied, namely that $t = sect1$ and $l_{in} \leq now$, then $g$ = down and $c$ = state2.

Proof. From Claim 5, if $t = sect1$, then $g$ = up and $c$ = state1, $g$ = lowering and $c$ = state2, or $g$ = down and $c$ = state2.

If $g$ = up and $c$ = state1, then from Claim 3, the assumption that $d2 + d3 < d1$, and Lemma 1 (i.e. now $\leq u_{lower}$), now $< l_{in}$, which contradicts the premise that $l_{in} \leq now$.

If $g$ = lowering and $c$ = state2, then from Claim 4, the assumption that $d2 + d3 < d1$, and Lemma 1 (i.e. now $\leq u_{down}$), now $< l_{in}$, which contradicts the premise that $l_{in} \leq now$.

Therefore, if $t = sect1$ and $l_{in} \leq now$, then $g$ = down and $c$ = state2. □
Thus, we consider a state in which \( CV \) raise is sufficient to consider a state in which \( CV \) and evolved the reformulated computational models called \( D9/D8 \) claim in the same way.

**Proof.** We show that transition rule \( in \) preserves the claim. It is sufficient to consider a state in which \( in \) is effective and its timing constraint is satisfied. Thus, we consider a state in which \( t = sect1 \) and \( l_{in} \leq now \). From Claim 6, \( g = down \) and \( c = state2 \) in this state as well. The following proof score can be used to show that \( in \) preserves the claim.

```plaintext
open CROSSING
ops s s' : -> Sys .
  eq t(s) = sect1 .
  eq g(s) = down .
  eq c(s) = state2 .
  eq t(s) <= now(s) = true .
  eq s' = in(s) .
  red t(s') == crossing and g(s') == down
    and c(s') == state2 .
```

We show that transition rule \( raise \) preserves the claim. It is sufficient to consider a state in which \( raise \) is effective. Thus, we consider a state in which \( g = down \) and \( c = state3 \). From the hypothesis, \( t \neq crossing \) in this state. The following proof score can be used to show that \( raise \) preserves the claim.

```plaintext
open CROSSING
ops s s' : -> Sys .
-- t(s) /\ crossing .
  eq g(s) = down .
  eq c(s) = state3 .
  eq s' = raise(s) .
  red t(s') == t(s) and g(s') == raising
    and c(s') == state0 .
```

We can show that any other transition rule preserves the claim in the same way. \( \square \)

**Claim 8.** In any reachable state, \( g = down \) whenever \( t = crossing \).

**Proof.** The claim immediately follows from Claim 7. \( \square \)

5. Concluding remarks

We have reformulated UNITY computational models and evolved the reformulated computational models called BCMs into TBCMs by introducing so-called clock variables so as to model real-time systems. Then, we have modeled the railroad crossing system as a TBCM, specified the TBCM modeling the system, and verified that the system has a safety property based on the specification with the help of the CafeOBJ system. The case study can be considered as a first step to expanding the application areas in which CafeOBJ can be used well.

It is well-known that computational models based on transition systems can be applied to modeling real-time systems by introducing clock variables. Among representative examples are TLA [1], timed automata [2], clocked transition systems [7, 8] and partially synchronous system models by I/O automata [10]. The definition of TBCMs is mainly affected by specification and verification of real-time systems with TLA [1] and with I/O automata [10]. How to describe TBCMs in CafeOBJ is similar to general timed automata described in [10].

The railroad crossing system has been widely used to demonstrate that formal methods can reason about timing-based systems. Our modeling is similar to that presented in [2].

We are going to specify more complicated real-time systems in CafeOBJ in the same manner so as to show that CafeOBJ can be applied to problems on a practical scale, and/or to find something to modify and/or add to CafeOBJ so that CafeOBJ can be more appropriately applied to such problems. In this paper, only simple safety properties of real-time systems have been treated. We are also going to deal with other properties such as bounded response and minimal separation [7, 8].

References