PAPER Special Section on Papers Selected from ITC-CSCC 2000

Specification and Verification of a Single-Track Railroad Signaling in CafeOBJ*

Takahiro SEINO[†a)], **Kazuhiro OGATA**[†], *and* **Kokichi FUTATSUGI**[†], ***Nonmembers***

SUMMARY A signaling system for a single-track railroad has been specified in CafeOBJ. In this paper, we describe the specification of arbitrary two adjacent stations connected by a single line that is called a *two-station system*. The system consists of two stations, a railroad line (between the stations) that is also divided into some contiguous sections, signals and trains. Each object has been specified in terms of their behavior, and by composing the specifications with projection operations the whole specification has been described. A safety property that more than one train never enter a same section simultaneously has also been verified with CafeOBJ.

key words: *CafeOBJ, formal methods, railroad signaling*

1. Introduction

Since key industrial systems such as railroad signaling systems and aviation control systems heavily affect people's lives, we must improve their safety as much as possible. We do not think that we can improve their safety in an ad hoc way because the systems are complex as well as huge. It is one possible approach to improving their safety that we formally specify the systems and verify that the systems have some desired properties based on the formal specifications.

Formal specification languages in which we can formally specify systems and with which we can formally verify that the systems have some properties have been proposed. CafeOBJ [4] is one of them. CafeOBJ allows us to specify state machines or objects of object-orientation in terms of their behavior.

We believe that case studies that we formally specify and verify some systems have to be done so that we can improve specification and verification techniques with formal specification languages such as CafeOBJ, and also make the languages easier to use. Therefore, as a case study we have done the following experiment. We have specified a kind of railroad signaling systems in CafeOBJ, and have formally verified that the system has an important safety property based on the formal specification with the help of the CafeOBJ system.

Railroad systems usually adopt block systems so as to prevent collisions between trains [9]. In block systems, railroad lines are partitioned into contiguous sections, in each of which at most one train is allowed to be. Railroad signaling systems are designed to aim at (semi-)automatically implementing block systems. We have dealt with a single-track railroad system that consists of a straight line on which more than one station are located. In this paper, we describe the specification of arbitrary two adjacent stations connected by a single line that is called a *two-station system* and the verification that no collision occurs.

The rest of the paper is organized as follows. Section 2 mentions CafeOBJ and how to specify systems in CafeOBJ and verify that the systems have some properties with CafeOBJ. Section 3 describes the two-station system, its specification in CafeOBJ, and the verification with CafeOBJ that the system has a safety property that more than one train never enter a same section simultaneously. In Sect. 4, we introduce some related works, and we finally conclude the paper in Sect. 5.

2. CafeOBJ in a Nutshell

CafeOBJ [4] is a direct successor of OBJ3 [7] that is one of the best-known algebraic specification languages. One of the outstanding features of CafeOBJ is that we can specify state machines or objects of object-orientation naturally, which were supposed to be difficult to specify in algebraic specification languages. The point is hidden algebra [6], with which we specify objects in terms of their behavior. There are two kinds of sorts in hidden algebra: *hidden* and *visible* sorts. A hidden sort represents the state space of an object, and a visible one usual data such as integers. There are also two kinds of operations: *action* and *observation* operations. An action operation may change the state of an object, and the state of an object can be only observed with observation ones. In addition, components are synthesized according to the component-based specification in CafeOBJ [5]. We use *projection* operations to combine specifications of component systems and build a specification of a composite system.

2.1 How to Specify Concurrent Systems in CafeOBJ

We show a specification for *fields of radio-buttons* as an example. Figure 1 shows a field of radio-buttons consisting of three buttons. We can use fields of radio-buttons to exclusively choose one among the buttons. A

Manuscript received October 2, 2000.
Manuscript revised December 22, 2000.

[†]The authors are with the Graduate School of Information Science, JAIST, Ishikawa-ken, 923-1292 Japan.
a) E-mail: t-seino@jaist.ac.jp

*This paper was presented at ITC-CSCC 2000.

Fig. 1 Field of radio-buttons.

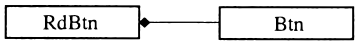


Fig. 2 UML object diagram for fields of radio-buttons.

UML object diagram for fields of radio-buttons is shown in Fig. 2. We first write a specification of buttons (corresponding to `Btn` in Fig. 2), and then we use a projection operation to combine the specification of buttons and build a specification for fields of radio-buttons (corresponding to `RdBtn` in Fig. 2).

We show the signature of a specification of buttons from which fields of radio-buttons are made:

```
op   init  : Bool-> Btn -- initial state.
bops on off: Btn -> Btn -- actions.
bop  on?   : Btn -> Bool-- observation.
```

A comment starts with `--` and terminates at the end of the line. `Btn` is a hidden sort representing the state space of each button, and `Bool` is a (built-in) visible sort representing boolean values. Operator `init` takes a boolean value, representing an initial state of a button. Action operators `on` and `off` can select and disselect a button, respectively. Observation operator `on?` allows us to observe the state of a button, i.e. selected or disselected. If a button is selected, `on?` returns `true`, and otherwise `on?` returns `false`. We use equations to define what happens next after applying an action operation to a button. The equations for buttons are as follows:

```
eq on? (init (B)) = B .
eq on? (on (S))  = true .
eq on? (off (S)) = false .
```

`B` and `S` are variables whose sorts are `Bool` and `Btn`, respectively. The first equation means that the initial state of a button is what is given to the button as its argument. The second (or third) equation means that the state of a button is changed to `true` (or `false`), i.e. selected (or disselected), after applying `on` (or `off`) to the button.

We show the signature of a specification of fields of radio-buttons:

```
-- initial state.
op  init : BtnID -> RdBtn
-- action.
bop on   : BtnID RdBtn -> RdBtn
-- observation.
bop on?  : BtnID RdBtn -> Bool
-- projection.
op  btn  : BtnID RdBtn -> Btn
```

Hidden sort `RdBtn` and visible one `BtnID` represent the state space of fields of radio-buttons and IDs for buttons, respectively. Projection operator `btn` is used to make fields of radio-buttons by combining buttons as components. More precisely, given a field of radio-buttons and an ID for a button in the field, `btn` takes out the corresponding button out from the field, or projects the field onto (the axis of) the button.

Operator `init` takes an ID as its argument, representing an initial state of a field of radio-buttons at which the button corresponding to the ID is selected and any other button is disselected. The following two (conditional) equations define this. Note that `BTN` and `BTN'` are variables whose sort is `BtnID`, and `R` is a variable whose sorts are `RdBtn`.

```
ceq btn (BTN, init (BTN')) = init (true)
    if BTN == BTN' .
ceq btn (BTN, init (BTN')) = init (false)
    if BTN =/= BTN' .
```

`init` on the left-hand sides is an operator on sort `RDBtn`, while `init` on the right-hand sides is one on sort `Btn`.

Action operator `on` chooses one among the buttons, namely that it makes the chosen button selected and makes any other button disselected. The following two equations define this.

```
ceq btn (BTN, on (BTN', R)) =
    on (btn (BTN, R))  if BTN == BTN' .
ceq btn (BTN, on (BTN', R)) =
    off (btn (BTN, R)) if BTN =/= BTN' .
```

`on` on the left-hand sides is an action operator on sort `RDBtn`, while `on` on the right-hand sides is one on sort `Btn`.

Observation operator `on?` observes the state of a button given by its first argument in a field of radio-buttons given by its second argument. The following equation defines this.

```
eq on? (BTN, R) = on? (btn (BTN, R)) .
```

`on?` on the left-hand side is an observation operator on sort `RDBtn`, while `on?` on the right-hand side is one on sort `Btn`.

Let us consider the following expression, or term:

```
on (1, on (3, on (2, init (1))))
```

where natural numbers are used to identify buttons. The term represents the state of a field of radio-buttons after some action operations are applied to the field. The initial state is that button `1` is selected and any other button is disselected, followed by choosing button `2`, button `3`, and again button `1`. Let `r` be the term. By applying `on?` to `r`, we can observe the state of each button in that state. For example, `on? (1, r)` is `true` and `on? (2, r)` is `false`.

2.2 How to Verify Concurrent Systems with CafeOBJ

We describe the verification of the claim that a field

of radio-buttons has the safety property that only one button is always selected. If a field of radio-buttons has one button, it is easy to show the claim. Hence, we suppose that there are at least two buttons in a field of radio-buttons. Since one and only button is initially selected in a field of radio-buttons, the claim is initially true. Hence, all we have to do is that given any state of a field of radio-buttons in which the claim holds, we show that the claim keeps holding in each next state after applying any action operation to the state. Let `rb` be a state where the claim holds, namely that only one button is selected. Since there is one action operation in a field of radio-buttons, we examine whether the claim keeps holding after applying the action operation to `rb`. We should consider two cases: (1) the selected button is selected again, and (2) any disselected button is selected. Let `b1` and `b2` be the selected button and an arbitrary disselected button in `rb`, respectively, and let `rb1` and `rb2` be the next states after selecting `b1` and `b2`, respectively. The case (2) is also divided into two cases that there are two buttons, and more than two buttons in `rb`. Let `b3` be an arbitrary disselected button except for `b2` in `rb` if there are more than two buttons in `rb`. The following proof score makes it possible to show that the claim keeps holding in `rb1` and `rb2`.

```
ops rb rb1 rb2 :-> RdBtn .
ops b1 b2 b3 :-> BtnID .
eq on? (btn(b1, rb)) = true .
eq on? (btn(b2, rb)) = false .
eq on? (btn(b3, rb)) = false .
eq rb1 = on (b1, rb) .
eq rb2 = on (b2, rb) .
red     on? (b1, rb1) == true
    and on? (b2, rb1) == false .
red     on? (b1, rb2) == false
    and on? (b2, rb2) == true
    and on? (b3, rb2) == false .
```

The equations on the third, fourth, and fifth lines mean that button `b1` is selected, and `b2` and `b3` representing an arbitrary disselected button are disselected in `rb`. The equations on the sixth and seventh lines define `rb1` and `rb2` as the next states after selecting `b1` and `b2`, respectively. CafeOBJ command `red` reduces a term by regarding given equations as left-to-right rewrite rules. The term following the first (or second) `red` means that the claim is also true in `rb1` (or `rb2`) as well. Both terms are reduced to `true`. We have completed the verification that a field of radio-buttons has the safety property.

3. A Single-Track Railroad System

We consider a two-station system shown in Fig. 3. The system has seven sections† $T_n\,(n = 1, \ldots, 7)$ and four signals $S_n\,(n = 1, \ldots, 4)$. A station consists of three sections: T_1, T_2, and T_3 for station A, and T_5, T_6, and T_7 for station B. A section has two properties: the number of trains in it and the direction. The direction has three possible values: L_{dir} (for left), R_{dir} (for right), and N_{dir} (for neutral). A signal has two possible states: G (for green) and R (for red) with usual meanings.

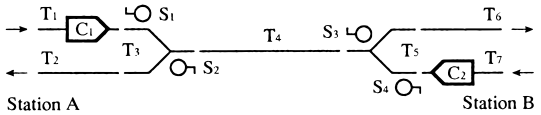


Fig. 3 Two-station system.

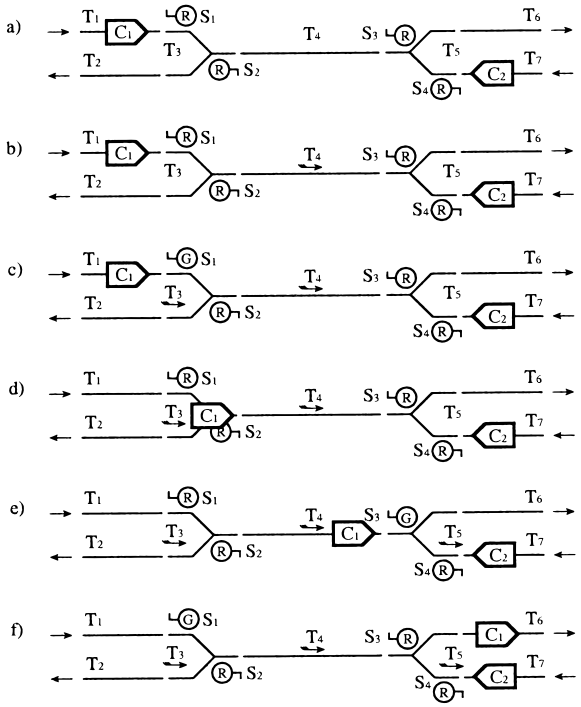


Fig. 4 One possible scenario.

Initially there are two trains C_1 and C_2 in the system as shown in Fig. 3, and every signal shows R. Besides, T_1 and T_6, T_2 and T_7, and T_3, T_4, and T_5 have R_{dir}, L_{dir}, and N_{dir}, respectively, in the initial state, and the directions of T_1, T_2, T_6, and T_7 cannot be changed.

Let us show one possible scenario that train C_1 reaches station B shown in Fig. 4:

1. Figure 4(a) shows the initial state.
2. It is confirmed whether the direction of T_4 is N_{dir}, and only if so, the direction is set to R_{dir} (see Fig. 4(b)).
3. It is confirmed whether the directions of T_3 and T_4 are N_{dir} and R_{dir}, respectively, and only if so, the direction of T_3 is set to R_{dir}. It is confirmed whether both directions of T_3 and T_4 are R_{dir}, and there is no train on T_3 and T_4, and only if so, S_1

†Each T_n may not actually correspond to a section, but in this paper it is regarded as a section for brevity.

is changed to G from R (see Fig. 4(c)).

4. It is confirmed whether S_1 is G, and only if so, C_1 is moved to T_3 from T_1 and S_1 is changed to R at the same time (see Fig. 4(d)), and then C_1 is moved to T_4.
5. It is confirmed whether the direction of T_5 is N_{dir}, and only if so, it is set to R_{dir}. It is confirmed whether the direction of T_5 is R_{dir}, and there is no train on T_5 and T_6, and only if so, S_3 is changed to G from R (see Fig. 4(e)).
6. It is confirmed whether S_3 is G, and only if so, C_1 is moved to T_5 from T_4 and S_3 is changed to R at the same time, and then C_1 is moved to T_6 (see Fig. 4(f)).

In the above scenario, we have mentioned how objects such as S_1 change their states. We describe how to change the states of objects in more detail.

- The direction of T_4 can be set to either R_{dir} or L_{dir} only if it is N_{dir}. It can be set back to N_{dir} from R_{dir} (or L_{dir}) if the direction of T_3 (or T_5) is N_{dir}.
- The direction of T_3 can be set to R_{dir} (or L_{dir}) only if it is N_{dir} and the direction of T_4 is R_{dir} (or any value). It can be set back to N_{dir} only if there is no train on it. The direction of T_5 can be changed likewise.
- S_1 can be changed to G from R only if there is no train on both T_3 and T_4, and both direction of T_3 and T_4 are R_{dir}. If a train enters T_3, or the direction of T_3 is set back to N_{dir}, S_1 must be set back to R simultaneously. S_4 can be changed likewise.
- S_3 can be changed to G from R only if there is no train on both T_5 and T_6, and the direction of T_5 is R_{dir}. If a train enters T_5, or the direction of T_5 is set back to N_{dir}, S_3 must be set back to R simultaneously. S_2 can be changed likewise.

3.1 Specification

We describe the specification of the two-station system in CafeOBJ. As the specification of fields of radio-buttons, specifications of components, i.e. trains and sections, are first written, and then the specification of the two-station system is built by combining the specifications of the components. Signals are represented in terms of sections. For example, S_1 is represented by the states of T_3 and T_4. If both directions of T_3 and T_4 are R_{dir}, and there is no train on both T_3 and T_4, then this case means that S_1 is G, and otherwise the other cases mean that S_1 is R. Figure 5 shows the UML object diagram corresponding to our specification.

The signature of the specification of trains is as follows:

```
-- initial state.
op   init-tr          : Dir -> Train
-- observations.
bop  dir?             : Train -> Dir
bop  where?           : Train -> TcID
-- actions.
bops move reach leave : Train -> Train
```

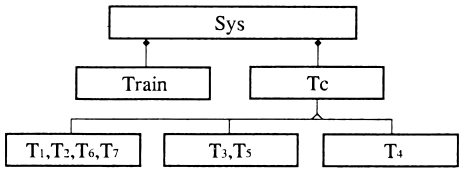


Fig. 5 UML object diagram for two-station systems.

`Train` is a hidden sort representing the state space of trains, and `Dir` and `TcID` are visible sorts representing directions to which train are running and IDs of sections, respectively. There are basically two kinds of trains in the two-station system. One is running right, and another running left. Hence, operator `init-tr` takes as its argument right or left, representing the initial state of a train. We have two observation operators: `dir?` and `where?`. `dir?` (or `where?`) takes as its argument (a state of) a train, and returns the direction to which the train is running (or the section on which the train is). We have three action operators: `move`, `reach`, and `leave`. `move` takes as its argument a train within the two-station system, and moves it to the next section (the right or left section depending on the direction to which the train is running) if a condition described above is true. `reach` takes as its argument a train out of the two-station system, and puts it on T_1 or T_7. A train running right (or left) is put on T_1 (or T_7). This action operation may be considered as a special version of `move`, which moves a train running right (or left) on the left (or right) section or yard of T_1 (or T_7) to T_1 (or T_7). `leave` is the opposite one that removes a train from T_1 (or T_7). The behavior of these observation and action operations are defined in equations in the same way as in fields of radio-buttons. Hence, we omit the definitions.

The specification of sections can be written in the same way as trains. In this paper, we omit the specification.

We show the main part of the specification of the two-station system:

```
-- initial state.
op  init   : -> Sys
-- observations.
bop watch? : SignalID Sys -> Signal
bop where? : TrainID Sys -> TcID
-- actions.
bop reach  : TrainID Sys -> Sys
bop leave  : TrainID Sys -> Sys
bop move   : TrainID Sys -> Sys
bop setdir : TcID Dir Sys -> Sys
```

```
-- projections.
op  train  : TrainID Sys -> Train
op  tc     : TcID Sys -> Tc
```

`Sys` is a hidden sort representing the state space of the two-station system, and `Train` and `Tc` also hidden sorts representing the state spaces of a train and a section that are components of the system. The other sorts are visible ones. `Bool` represents the boolean values, `TrainID`, `SignalID`, and `TcID` represent IDs of trains, signals, and sections, respectively, and `Signal` and `Dir` represent values of signals and directions of sections, respectively.

Operator `init` represents the initial state of the two-station system. Operators `watch?` and `where?` are observation ones. `watch?` returns either R or G of the signal given as its first argument. `where?` returns the section where the train given as its first argument is. Operator `reach`, `leave`, `move`, and `setdir` are action ones. `reach` puts a train running right (or left) on T_1 (or T_7), which means that a train enters a station from a yard or the previous section of T_1 (or T_7). `leave` is the opposite one that removes a train from T_1 (or T_7). `move` moves a train to the next section. If the next section has a signal, the operator is enabled (or can change the system state) only if the signal is G. `setdir` sets a section (except for T_1, T_2, T_6, and T_7) to either L_{dir}, R_{dir}, or N_{dir}. Operators `train` and `tc` are projection ones that combine the specifications of trains and sections to build the specification of the two-station system.

We describe how to define each operation in equations.

Action operator `setdir` only affects each section T_n in the two-station system. Each train C_n cannot be affected by `setdir` at all. So, it is very simple to define `setdir` for projection operator `train` as follows:

```
eq train (TR, setdir (TC, D, S)) =
                          train (TR, S) .
```

The equation means that even if `setdir` sets section `TC` in system `S` to direction `D`, train `TR` does not change its state at all. On the other hand, `setdir` for projection operator `tc` is defined as follows:

```
ceq tc (TC, setdir (TC', D, S)) =
                  setdir (D, tc (TC, S))
    if  TC == TC'
    and setdir-cond (TC, D, S) .
ceq tc (TC, setdir (TC', D, S)) =
                  tc (TC, S)
    if  TC =/= TC'
    or  not (setdir-cond (TC, D, S)) .
```

`setdir` on the left-hand side of each equation is an action operator on `Sys`, and `setdir` on the right-hand side is an action operator on `Tc`. The first equation means that if `setdir` tries to set section `TC'` in system `S` to direction `D` provided that condition `setdir-cond` is satisfied, section `TC'` is actually set to the direction. The second equation means that even if `setdir` tries to set `TC'` in `S` to `D`, any other section `TC` does not change its state, and section `TC'` does not change its state either unless condition `setdir-cond` is satisfied.

Condition `setdir-cond` is defined for each section T_n. For sections `t1`, `t2`, `t6`, `t7`, and `yard`, condition `setdir-cond` is always false as defined as follows:

```
op setdir-cond : TcID Dir Sys -> Bool
eq setdir-cond (t1, D, S) = false .
eq setdir-cond (t2, D, S) = false .
eq setdir-cond (t6, D, S) = false .
eq setdir-cond (t7, D, S) = false .
eq setdir-cond (yard, D, S) = false .
```

where `t`n and `yard` are constants representing T_n and the previous section of either T_1 or T_7, respectively. For `t3`, `t4`, and `t5`, condition `setdir-cond` is defined as described earlier. The definition is as follows:

```
eq setdir-cond (t3, L, S) =
                  dir? (tc (t3, S)) == N .
eq setdir-cond (t3, R, S) =
                  dir? (tc (t3, S)) == N
              and  dir? (tc (t4, S)) == R .
eq setdir-cond (t3, N, S) =
              not  (exist? (tc (t3, S))) .
eq setdir-cond (t4, L, S) =
                  dir? (tc (t4, S)) == N .
eq setdir-cond (t4, R, S) =
                  dir? (tc (t4, S)) == N .
eq setdir-cond (t4, N, S) =
              (dir? (tc (t4, S)) == R
              and  dir? (tc (t3, S)) == N)
              or  (dir? (tc (t4, S)) == L
              and  dir? (tc (t5, S)) == N) .
eq setdir-cond (t5, L, S) =
                  dir? (tc (t5, S)) == N
              and  dir? (tc (t4, S)) == L .
eq setdir-cond (t5, R, S) =
                  dir? (tc (t5, S)) == N .
eq setdir-cond (t5, N, S) =
              not (exist? (tc (t5, S))) .
```

where constants `L`, `R`, and `N` represent L_{dir}, R_{dir}, and N_{dir}, respectively, and `dir?` and `exist?` are observation operators on `Tc` with which we can observe the direction of each section and confirm whether there exist trains on each section, respectively. For example, for section `t3` in system `S` and direction `L`, condition `setdir-cond` is true if the direction of `t3` in `S` is `N`.

Observation operator `watch?` obtaining the state of each signal is defined as follows:

```
ceq watch? (SG, S) = G
    if signal-cond (SG, S) .
ceq watch? (SG, S) = R
    if not (signal-cond (SG, S)) .
```

Signal `SG` is `G` (or `R`) if condition `signal-cond` is satisfied (or not). Condition `signal-cond` is defined for each signal as follows:

```
op signal-cond : SignalID Sys -> Bool
eq signal-cond (s1, S) =
            exist? (tc (t3, S)) == false
       and  exist? (tc (t4, S)) == false
       and  dir? (tc (t3, S)) == R .
eq signal-cond (s2, S) =
            exist? (tc (t2, S)) == false
       and  exist? (tc (t3, S)) == false
       and  dir? (tc (t3, S)) == L .
eq signal-cond (s3, S) =
            exist? (tc (t5, S)) == false
       and  exist? (tc (t6, S)) == false
       and  dir? (tc (t5, S)) == R .
eq signal-cond (s4, S) =
            exist? (tc (t4, S)) == false
       and  exist? (tc (t5, S)) == false
       and  dir? (tc (t5, S)) == L .
```

where `s`n is a constant representing S_n. The above equations basically correspond to what we have described on behavior of each signal except that the direction of T_4 is not inspected. The reason why the inspection does not need is because if the direction of T_3 (or T_5) is R_{dir} (or L_{dir}), it is clear from the definition of `setdir-cond` that the direction of T_4 is also R_{dir} (or L_{dir}).

Observation operation `where?` is defined simply as follows:

```
eq where? (TR, S)
  = where? (train (TR, S)) .
```

`where?` on the left-hand side is an observation operator on `Sys`, while `where?` on the right-hand side is one on `Train`.

Action operator `move` for projection operator `train` is defined as follows:

```
ceq train (TR, move (TR', S)) =
   move (train (TR, S))
   if  TR == TR'
   and move-cond (where? (TR, S), TR, S).
ceq train (TR, move (TR', S)) =
   train (TR, S)
   if  TR =/= TR'
   or not
   (move-cond (where? (TR, S), TR, S)).
```

`move` on the left-hand side of each equation is an action operator on `Sys`, and `move` on the right-hand side is an action operator on `Train`. The first equation means that if `move` tries to move train `TR'` to the next section provided that condition `move-cond` is satisfied, train `TR'` is actually moved to the next section. The second equation means that even if `move` tries to move train `TR'` to the next section, any other train does not move at all, and train `TR'` does not move either unless `move-cond` is satisfied. Action operator `move` for projection operator `tc` is defined as follows:

```
ceq tc (TC, move (TR, S)) =
  enter (tc (TC, S))
  if  TC == where? (move (train (TR, S)))
and  move-cond (where? (TR, S), TR, S) .
ceq tc (TC, move (TR, S)) =
  exit (tc (TC, S))
  if  TC == where? (train (TR, S))
  and  move-cond (
                 where? (TR, S), TR, S) .
ceq tc (TC, move (TR, S)) = tc (TC, S)
  if  TC =/= where? (train (TR, S))
  or  TC =/=
            where? (move (train (TR, S)))
  or not
     (move-cond
               (where? (TR, S), TR, S)) .
```

where `enter` (or `exit`) is an action operator on `Tc`, meaning that a train has entered (or exited) the section, and `where?` is an observation operator on `Train` observing the section on which there exists the train. The first (or second) equation means that if `move` tries to move train `TR` in system `S` provided that condition `move-cond` is satisfied, train `TR` enters the next of the section where `TR` is (or exits the section where `TR` is). The third equation means that even if `move` tries to move `TR` in `S`, no train enters and/or exits any other section, and no train enters and/or exits the section where `TR` is and the next section unless condition `move-cond` is satisfied.

Condition `move-cond` is defined for each section as follows:

```
op move-cond : TcID TrainID Sys -> Bool
eq move-cond (t1, TRR, S) =
                  watch? (s1, S) == G .
eq move-cond (t2, TRR, S) = false .
eq move-cond (t3, TRR, S) = true .
eq move-cond (t4, TRR, S) =
                  watch? (s3, S) == G .
eq move-cond (t5, TRR, S) = true .
eq move-cond (t6, TRR, S) = false .
eq move-cond (t7, TRR, S) = false .
eq move-cond (yard, TRR, S) = false .
```

The above equations are good for a train running right. For example, a train on section T_1 can move to section T_3 if signal S_1 shows G, a train on section T_2 cannot move to section T_3 at any time, and a train on section T_3 can always move to section T_4, which are represented by the first, second, and third equations, respectively. The equations for a train running left can be defined as well. Note that as described above, action operator `move` does not move a train on section T_6 to the yard or the right section of T_6, but action operator

`leave` does this. Action operators `reach` and `leave` may be considered as special versions of `move`. They can be defined as `move`.

3.2 Verification

We have proved that the two-station system has the safety property that more than one train never enter a same section simultaneously. We describe the verification.

Basically we have used the same verification technique described in Sect. 2. In the two-station system, however, there are states such that although the states have the property, the property is not preserved in the next states after applying some action operation to the states. Therefore, we first find out such states, and then show that these states are not reachable from the initial state.

There are basically four cases corresponding to such cases. For the symmetry of the two-station system, however, only two cases should be considered. The two cases are (r1) and (r2) shown in Fig. 6.

First let us consider the case (r1). Suppose that there exist two trains moving left on T_2 and T_3, respectively, the two trains are on T_2 simultaneously if action operator `move` is applied to the train on T_3. Now we show that any state corresponding to the case (r1) is not reachable. Although there are more than one state that are predecessors of the states corresponding to the case (r1), we only need to consider the states corresponding to the case (r1') because any other previous state coincides with one of the states corresponding to the case (r1). Only applying `move` to the train on T_4 in the case (r1') could change a state corresponding to (r1') to a state corresponding to (r1). Therefore, we have only to show that such a transition cannot be happened. The following proof score can prove this:

```
ops c1 c2 : -> TrainID .
ops r1 r1' : -> Sys .
eq  where? (train (c1, r1')) = t2 .
eq  where? (train (c2, r1')) = t4 .
eq  dir? (train (c1, r1')) = L .
eq  dir? (train (c2, r1')) = L .
```

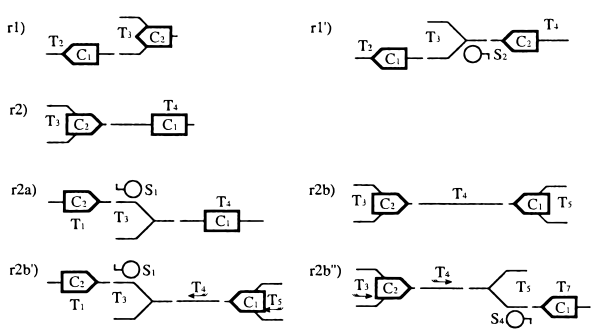


Fig. 6 Unsafe but unreachable states.

```
eq  exist? (tc (t2, r1')) = true .
eq  exist? (tc (t3, r1')) = false .
eq  exist? (tc (t4, r1')) = true .
eq  r1 = move (c2, r1') .
red where? (c2, r1') == where? (c2, r1) .
```

Constants `c1` and `c2` are IDs of two trains in the case (r1') of Fig. 6. Constant `r1'` represents a state corresponding the case (r1') of Fig. 6. State `r1'` is characterized with above equations. For example, the equation on the third line means that train `c1` is on section `t2` in state `r1'`. In the proof, we do not define any other components such as T_5 that do not matter in the proof. Constant `r1` represents the state after trying to apply action operation `move` to train `c2` in state `r1'`. The term following `red` means that the section on which train `c2` is in state `r1` is the same as in state `r1'`, namely that any state corresponding to the case (r1) is not reachable from the initial state. The term has been reduced to `true`.

Next let us consider the case (r2). Suppose that there exist a train moving right on T_3 and a train moving either left or right on T_4, the two trains are on T_4 simultaneously if action operator `move` is applied to the train on T_3. We can show that any state corresponding to the case (r2) is not reachable in the same way as the case (r1). In this case, there are two cases (r2a) and (r2b) corresponding to the states that are predecessors of the states corresponding to the case (r2). Moreover, we have to consider two cases (r2b') and (r2b") that are predecessors of the states corresponding to the case (r2b) because a state corresponding to the case (r2b) can be changed to a state corresponding to the case (r2). That is, all that is needed is to show that any state corresponding to one of the three cases (r2a), (r2b') and (r2b") does not lead to any state corresponding to the case (r2). In this paper, we only show that any state corresponding to the case (r2b') does not lead to any state corresponding to the case (r2b), which implies that it does not lead to any state corresponding to the case (r2). The following proof score makes it possible to show this:

```
op  c1  : -> TrainID .
op  c2  : -> TrainID .
ops r2b r2b' : -> Sys .
eq  where? (train (c1, r2b')) = t5 .
eq  where? (train (c2, r2b')) = t1 .
eq  dir? (train (c2, r2b')) = R .
eq  exist? (tc (t1, r2b')) = true .
eq  exist? (tc (t3, r2b')) = false .
eq  exist? (tc (t4, r2b')) = false .
eq  exist? (tc (t5, r2b')) = true .
eq  dir? (tc (t4, r2b')) = L .
eq  dir? (tc (t5, r2b')) = L .
eq  r2b = move (c2, r2b') .
red where? (c2, r2b')
    == where? (c2, r2b') .
```

The other two cases can be done likewise.

We have completed the verification that the two-station system has the safety property.

4. Related Work

Block systems are the principal concept for safety assurance on the railroad domain. Cichoki and Gorski describe a formal specification of railroad signaling systems written in Z and show some safety properties and hazards on the systems with FMEA (Failure Mode and Effect Analysis) analysis technique [3]. The technique is a kind of methodology for system analysis with bottom up approach, and aims to identify and document anticipated faults of the components and their impact on the system external interfaces. Cichoki and Gorski indicate that some hazards cause failures of the lowest components (hardware) on their model, and show countermeasures for some cases on the hazards.

In the railroad domain, to synthesize signals and branches are called *interlocking*, and each station needs an interlocking controller. There are many works of applying formal methods to interlocking design. For example, Morley models interlocking logic with higher order logic and implements his model and a model checker in Standard ML [8]. He designs a special language with which we can specify rails, signals and branches on a station and prove full-automatically some safety properties about interlocking with the models and the model checker. But it is still difficult to prove properties interlocking for huge stations.

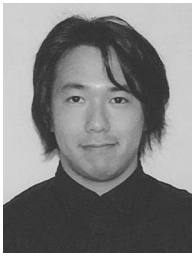
Bjørner et al. model many functions in the railroad domain and describe their requirements as widely as possible. They aim to illustrate what a railroad system is by decomposing the system into a number of components. The domain models and requirement definitions are written both informally in English and formally in the RAISE Specification Language [1], [2]. Their works are still in progress, and these papers are incomplete.

5. Conclusion

We have described the specification of a single-track railroad system in CafeOBJ, and the verification of its signaling system that no collision between trains occurs if trains run according to the signals with the help of the CafeOBJ system.

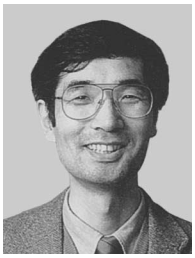
References

[1] D. Bjørner, J. Braad, and K. Mogensen, "Models of railway systems: Domain," Proc. 5th Workshop on FMERAIL, 1999.
[2] D. Bjørner, J. Braad, and K. Mogensen, "Models of railway systems: Requirements," Proc. 5th Workshop on FMERAIL, 1999.
[3] T. Cichocki and J. Gorski, "Safety assessment of computerized railway signaling equipment supprted by formal techniques," Proc. 5th Workshop on FMERAIL, 1999.
[4] R. Diaconescu and K. Futatsugi, CafeOBJ report, World Scientific, 1998.
[5] R. Diaconescu, K. Futatsugi, and S. Iida, "Component-based algebraic specification and verification in CafeOBJ," Proc. FM'99, LNCS 1709 Springer, pp.1644–1663, 1999.
[6] J. Goguen and G. Malcolm, "A hidden agenda," Theoretical Computer Science, 245, pp.55–101, 2000.
[7] J. Goguen and G. Malcom, "Software engineering with OBJ—Algebraic specification in action," Advances in formal methods, vol.2, Kluwer, 2000.
[8] M. Morley, "Safety assurance in interlocking design," Ph.D. Thesis of Univ. of Edinburgh, 1996.
[9] I. Yoshitake and Y. Akimoto, An explanation of driving security facilities, Japan Railway Books Inc., 1984.



Takahiro Seino received his Master of Information Science from JAIST in 1998. He is a Ph.D. candidate in School of Information Science, JAIST. His primary research interests are distributed algorithms and systems, and their formal specification and verification.

Kazuhiro Ogata received the Ph.D. degree in engineering from Keio University in 1995. He is a Research Associate in School of Information Science, JAIST. He joined JAIST in 1995. His primary research interests include distributed algorithms and systems, their formal specification and verification, and design and implementation of computer languages.



Kokichi Futatsugi received the Ph.D. degree in engineering from Tohoku University in 1975. He is a Professor in School of Information Science, JAIST. He joined ETL in 1975. He assumed a concurrent position at JAIST in April 1992 while he was working mainly for ETL as a Chief Senior Researcher. In April 1993, he started to work mainly for JAIST. His primary research goal is to design and develop new computer languages which can open up new application areas, and/or improve the current software technology.