Mechanically Supporting Case Analysis for Verification of Distributed Systems

Takahiro Seino
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, JAPAN
t-seino@jaist.ac.jp
Kazuhiro Ogata
NEC Software Hokuriku, Ltd.
1 Anyoji, Hakusan, Ishikawa 920-2141, JAPAN
ogatak@acm.org
and Kokichi Futatsugi
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, JAPAN
kokichi@jaist.ac.jp

Received: January XX 2005; revised: November XX 2005

***Abstract*— The OTS/CafeOBJ method can be used to formally model, specify and verify distributed systems such as security protocols and railroad systems. A distributed system is modeled as an OTS, a kind of transition system, and the OTS is specified and verified with CafeOBJ, an algebraic specification language. Case analysis (or case splitting) is one of the most intellectual pieces of work in verification. Case analysis should be done entirely by hand in the OTS/CafeOBJ method, which is error-prone. It is indispensable to cover all cases and find necessary lemmas for some sub-cases where desired results are not obtained in case analysis. We propose two methods of mechanically supporting case analysis, which concern these two issues. A case study that the proposed methods are effectively applied to a railroad signaling system is also reported.**

***Index Terms*— Algebraic specification languages, Distributed systems, Formal methods, Rewriting, Verification**

I. INTRODUCTION

In the advanced information society in the 21st century where the world wide network, namely the Internet, is the crucial infrastructure and computers pervade all scenes of our life, namely the pervasive/ubiquitous computing, it will be often the case that we should develop systems reliably offering high quality services at any expense. Systems that should be reliable are often reactive and/or distributed systems, which change their states from moment to moment and keep on offering services. Among such systems are electronic commerce systems and railroad control systems. When such systems are developed, it is essential to find and correct faults lurked in the designs and specifications as much as possible at earlier stages of the development process before coding them in programming languages.

One of the existing approaches to this issue is to formally model (the designs of) such systems and formally verify that they have desired properties. Distributed systems are often modeled as transition systems[2][12][13]. If a distributed system can be modeled as a finite-state transition system, model-checking techniques[5] must be one of the best choices. Otherwise, basically theorem-proving techniques should be used.

Many tools supporting theorem proving have been proposed such as Coq[1] and Isabell/HOL[14]. Our method called the OTS/CafeOBJ method[18] uses CafeOBJ[3], an algebraic specification language/system. The CafeOBJ's basic mechanism for theorem proving is rewriting, which is an efficient way of implementing equational reasoning. Equational reasoning is the most fundamental way of reasoning, which can moderate the difficulties of proofs that might otherwise become too hard to understand. Consequently, we believe that our method is easier to learn and use than those based on Coq and Isabell/HOL that rely on a type theory and a higher-order logic.

In the OTS/CafeOBJ method, a distributed system is modeled as an OTS (Observational Transition System), which is a transition system that can be appropriately written in equations. The OTS is written in equations using CafeOBJ. Proofs, or proof scores that the OTS has properties are written in CafeOBJ, while the proof scores are checked by means of rewriting with the CafeOBJ system. We have demonstrated the effectiveness of the OTS/CafeOBJ method by performing case studies that the method has been applied to electronic commerce protocols[17], distributed mutual exclusion algorithms[16], railroad signaling systems[20], real-time

This paper is a revised version of [21]. This research is partly conducted as a program for the ”Fostering Talent in Emergent Research Fields” in Special Coordination Funds for Promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology.

systems[15] and hybrid systems[19].

However, proof scores are basically written by hand, which may lead to human errors such that some cases to check may be overlooked. To minimize human errors, writing proof scores should be mechanically supported. In this paper, we focus on case analysis, which is one of the most intellectual pieces of work in verification. Case analysis in the OTS/CafeOBJ method involves finding appropriate predicates and splitting a case into multiple sub-cases based on the predicates. Let us consider a proof of $\forall t1, t2 \in \mathrm{TrID}.P(t1, t2)$ where TrID is a set of train IDs. To proceed with the proof, we may have to split the case into two sub-cases based on the predicate "$t1 = t2$", and reduce the proof to two sub-proofs: (1) $P(t1, t1)$ and (2) $P(t1, t2)$ where $t1 \neq t2$.

When case analysis is performed, it is indispensable to cover all cases and find necessary lemmas for some sub-cases where desired results are not obtained. We propose two methods of mechanically supporting case analysis, which concern these two issues. The first method uses matrices to cover all cases. The matrices consist of predicates that come from transitions' conditions and properties to prove. If it is not sufficient to split cases with such matrices for verification, we must decide to split some sub-cases furthermore or find necessary lemmas for some sub-cases. Given a set of basic predicates found in the specifications of OTSs modeling distributed systems, the second method mostly automates case analysis more precisely, which can help find necessary lemmas.

A case study has been performed to demonstrate the effectiveness of these two methods. In this case study, we verify that there are no collisions in a railroad where trains run according to the staff system (or the tablet blocking system) that is a railroad signaling system. The case study is also reported in this paper.

The rest of the paper is organized as follows: Section II mentions the OTS/CafeOBJ method. Section III describes the proposed methods. Section IV reports on the case study. Section V discusses some related work. Section VI concludes the paper.

II. The OTS/CafeOBJ Method

A. CafeOBJ: Algebraic Specification Language and System

CafeOBJ[1][3] is mainly based on initial algebras[7] and hidden algebras[4][9]. Abstract machines as well as abstract data types can be specified in CafeOBJ. There are two kinds of sorts in CafeOBJ, which are visible and hidden sorts. A visible sort denotes an abstract data type, while a hidden sort denotes the state space of an abstract machine. There are two kinds of operators (or operations) with respect to hidden sorts, which are action and observation operators. Action operators denote state transitions of abstract machines, while observation operators let us know the situation where abstract machines are located. Both an action operator and an observation operator take a state of an abstract machine and zero or more data. The action operator returns the successor state of the state with respect to the state transition denoted by the action operator plus the data. The observation operator returns a value that characterizes the situation where the abstract machine is located.

[1]See `http://www.ldl.jaist.ac.jp/cafeobj/`

Visible sorts are declared by enclosing `[` and `]`, and hidden sorts are declared by enclosing `*[` and `]*`. Action and observation operators are declared by starting with `bop`, and other operators are declared by starting with `op`. After `bop` or `op`, an operator name is written, followed by a colon `:` and a list of sorts, and then, `->` and a sort are written. The list of sorts is called the arity of the operator, and the sort after `->` is called the coarity of the operator. The pair of the arity and coarity is called the rank of the operator. When declaring more than one operator whose rank is the same simultaneously, `bops` and `ops` are used instead of `bop` and `op`. Operators with the empty arity are called constants.

Operators are defined in equations. An equation is declared by starting with `eq`, and a conditional equation is declared by starting with `ceq`. After `eq`, two terms connected with `=` are written, ended with a full stop. After `ceq`, two terms connected with `=` are written, followed by `if`, and then, a term denoting the condition and a full stop are written.

The CafeOBJ system uses declared equations as left-to-right rewrite rules and rewrites (or reduces) a given term. The command `red` is used to reduce a given term. This executability makes it possible to simulate a specified system and verify that a specified system has properties.

Basic units of CafeOBJ specifications are modules. The CafeOBJ system provides built-in modules where basic data types such as truth values are specified. The module of truth values is `BOOL`.

Since truth values are indispensable for conditional equations, `BOOL` is automatically imported by almost every module unless otherwise stated. The import of `BOOL` lets us use the visible sort `Bool` denoting truth values, the constants `true` and `false` denoting true and false, and operators denoting some basic logical operators. Among the operators are `not_`, `_and_`, `_or_`, `_xor_`, `_implies_` and `_iff_` denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), exclusive disjunction (xor), implication ($\Rightarrow$) and logical equivalence ($\Leftrightarrow$), respectively. The operator `if_then_else_fi` corresponding to **if** statements in programming languages is also available. An underscore _ indicates the place where an argument is put.

`BOOL` plays an essential role in verification with the CafeOBJ system. If the equations available in the module are regarded as left-to-right rewrite rules, they are complete with respect to propositional logic[11]. Therefore, any term denoting a propositional formula that is always true (or false) is surely reduced to `true` (or `false`). Generally, a term denoting a propositional formula is reduced to a term denoting an exclusively disjunctive normal form of the propositional formula.

B. Observational Transition Systems

We assume that there exists a universal state space denoted by Υ. We also assume that data types used, including the equivalence relation (denoted by $=$) for each data type, have been defined in advance. Let B and N be the set of truth values and the set of natural numbers, respectively.

Definition 1 (OTS): An OTS[18] (observational transition system) $\mathcal{S}$ consists of $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ where

- $\mathcal{O}$: A set of observers. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \to D$, where D is a data type and may differ from observer to observer. Given two states $v_1, v_2 \in \Upsilon$, the equivalence between the two states, denoted by $v_1 =_{\mathcal{S}} v_2$, with respect to $\mathcal{S}$ is defined as $\forall o \in \mathcal{O}. o(v_1) = o(v_2)$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A set of conditional transitions. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon \to \Upsilon$, provided that $\tau(v_1) =_{\mathcal{S}} \tau(v_2)$ for each $[v] \in \Upsilon / =_{\mathcal{S}}$ and each $v_1, v_2 \in [v]$. $\tau(v)$ is called the successor state of $v \in \Upsilon$ with respect to τ. The condition $c_\tau : \Upsilon \to B$ of $\tau \in \mathcal{T}$ is called *the effective condition*. τ is required to satisfy the requirement that $v =_{\mathcal{S}} \tau(v)$ for each $v \in \Upsilon$ such that $\neg c_\tau(v)$. □

Definition 2 (Execution): An execution of an OTS $\mathcal{S}$ is an infinite sequence $v_0, v_1, \ldots$ of states satisfying

- *Initiation*: $v_0 \in \mathcal{I}$.
- *Consecution*: For each $i \in N$, there exists $\tau \in \mathcal{T}$ such that $v_{i+1} =_{\mathcal{S}} \tau(v_i)$.

Let $\mathcal{E}_{\mathcal{S}}$ be the set of all executions obtained from $\mathcal{S}$. □

A state $v \in \Upsilon$ appears in an execution $v_0, v_1, \ldots$ of an OTS $\mathcal{S}$, denoted by $v \in v_0, v_1, \ldots$, if and only if there exists $i \in N$ such that $v =_{\mathcal{S}} v_i$.

Definition 3 (Reachable state): A state $v \in \Upsilon$ is called *reachable* with respect to an OTS $\mathcal{S}$ if and only if there exists an execution $e \in \mathcal{E}_{\mathcal{S}}$ such that $v \in e$. Let $\mathcal{R}_{\mathcal{S}}$ be the set of all reachable states with respect to $\mathcal{S}$. □

All properties considered in this paper are invariants.

Definition 4 (Invariant): A predicate $p : \Upsilon \to B$ is called invariant with respect to an OTS $\mathcal{S}$, denoted by $\mathsf{invariant}_{\mathcal{S}}\ p$, if and only if $\forall v \in \mathcal{R}_{\mathcal{S}}. p(v)$. $\mathcal{S}$ may be omitted from $\mathsf{invariant}_{\mathcal{S}}\ p$ if it is clear from context. □

Let $x_1, x_2, \ldots$, whose data types are $D_1, D_2, \ldots$, be all free variables in $\mathsf{invariant}_{\mathcal{S}}\ p$. We suppose that $\mathsf{invariant}_{\mathcal{S}}\ p$ is interpreted as $\forall x_1 \in D_1. \forall x_2 \in D_2 \ldots (\mathsf{invariant}_{\mathcal{S}}\ p)$ in this paper. When a proof score of this formula is written, the free variables are replaced with constants denoting arbitrary values of the corresponding data types and the universal quantifiers are eliminated.

Observers and transitions may be parameterized, which are generally expressed as $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, respectively, provided that $m, n \geq 0$ and there exists a data type D_k such that $k \in D_k$, where $k = i_1, \ldots, i_m, j_1, \ldots, j_n$. For example, an integer array a possessed by a process p may be denoted by an observer a_p, and the increment of the ith element of the array may be denoted by a transition $inc\text{-}a_{p,i}$.

C. Writing Observational Transition Systems

An OTS $\mathcal{S}$ is written in CafeOBJ. The universal state space Υ is denoted by a hidden sort, say H. An observer $o_{i_1,\ldots,i_m} \in \mathcal{O}$ is denoted by a CafeOBJ observation operator. We assume that there exist visible sorts V_k and V corresponding to the data types D_k and D, where $k = i_1, \ldots, i_m$. The CafeOBJ observation operator denoting $o_{i_1,\ldots,i_m}$ is declared as follows:

`bop` $o : V_{i_1} \ldots V_{i_m}\ H$ `->` V

Any state in $\mathcal{I}$, namely any initial state, is denoted by a constant, say *init*, which is declared as follows:

`op` *init* : `->` H

The initial value returned by $o_{i_1,\ldots,i_m}$ is denoted by the term $o(X_{i_1}, \ldots, X_{i_m}, init)$, where X_k is a CafeOBJ variable whose sort is V_k, where $k = i_1, \ldots, i_m$. The initial value can be generally specified as follows:

`eq` $P_1(o(X_{i_1}, \ldots, X_{i_m}, init), Y_{11}, \ldots, Y_{1M_1}) =$ `true .`
$\ldots$
`eq` $P_N(o(X_{i_1}, \ldots, X_{i_m}, init), Y_{N1}, \ldots, Y_{NM_N}) =$ `true .`

P_k is a predicate that specifies the initial value returned by $o_{i_1,\ldots,i_m}$, where $k = 1, \ldots, N$ and $N \geq 0$. Y_l is a CafeOBJ term (which may be a CafeOBJ variable), where $l = 11, \ldots, 1M_1, \ldots, N1, \ldots, NM_N$.

A transition $\tau_{j_1,\ldots,j_n} \in \mathcal{T}$ is denoted by a CafeOBJ action operator. We assume that there exists a visible sort V_k corresponding to the data type D_k, where $k = j_1, \ldots, j_n$. The CafeOBJ action operator denoting $\tau_{j_1,\ldots,j_n}$ is declared as follows:

`bop` $a : V_{j_1} \ldots V_{j_n}\ H$ `->` H

$\tau_{j_1,\ldots,j_n}$ may change the value returned by $o_{i_1,\ldots,i_m}$ if it is applied in a state v such that $c_{\tau_{j_1,\ldots,j_n}}(v)$, which can be written generally as follows:

`ceq` $o(X_{i_1}, \ldots, X_{i_m}, a(X_{j_1}, \ldots, X_{j_n}, S))$
    $= e\text{-}a(X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m}, S)$
      `if` $c\text{-}a(X_{j_1}, \ldots, X_{j_n}, S)$.

S is a CafeOBJ variable whose sort is H and X_k is a CafeOBJ variable whose sort is V_k, where $k = j_1, \ldots, i_n$. $a(X_{j_1}, \ldots, X_{j_n}, S)$ denotes the successor state of S with respect to $\tau_{j_1,\ldots,j_n}$ plus $X_{j_1}, \ldots, X_{j_n}$. $e\text{-}a(X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m}, S)$ denotes the value returned by $o_{i_1,\ldots,i_m}$ in the successor state. $c\text{-}a(X_{j_1}, \ldots, X_{j_n}, S)$ denotes the effective condition $c_{\tau_{j_1,\ldots,j_n}}$.

The value returned by $o_{i_1,\ldots,i_m}$ is not changed if $\tau_{j_1,\ldots,j_n}$ is applied in a state v such that $\neg c_{\tau_{j_1,\ldots,j_n}}(v)$, which can be written generally as follows:

`ceq` $o(X_{i_1}, \ldots, X_{i_m}, a(X_{j_1}, \ldots, X_{j_n}, S))$
    $= o(X_{i_1}, \ldots, X_{i_m}, S)$
      `if not` $c\text{-}a(X_{j_1}, \ldots, X_{j_n}, S)$.

The declaration and definition of the CafeOBJ action operator denoting $\tau_{j_1,\ldots,j_n}$ can also be written in a more succinct way as follows:

`act` $a : X_{j_1}{:}V_{j_1} \ldots X_{j_n}{:}V_{j_n}\ S{:}H$ `->` H .
    `cond` $c\text{-}a(X_{j_1}, \ldots, X_{j_n}, S)$.
    `eff` $o(X_{i_1}, \ldots, X_{i_m}, S')$
                `:=` $e\text{-}a(X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m}, S)$.
    `eff ... .`
    $\ldots$
`tca`

This is translated into the declaration of a and the equations defining a such as those described above. S' denotes the

```
op s : -> H .         -- s denotes an arbitrary state.
eq q(s) = true .      -- q holds in s.
eq r(s) = false .     -- r holds in s.
eq q(s) = true .      -- p holds in s, which is the I.H.
red p(a(s)) .         -- Check if a preserves p in s.
```

Fig. 1. A typical passage of proof scores in the OTS/CafeOBJ method.

successor state of S with respect to $\tau_{j_1,\ldots,j_n}$.

III. ORGANIZING CASE ANALYSIS

We first review proof scores written in the OTS/CafeOBJ method. We suppose that we prove a predicate $p : \Upsilon \to B$ invariant to an OTS $\mathcal{S}$, namely that we prove $\mathsf{invariant}_{\mathcal{S}}\ p$ by induction on the number of transitions applied. In an induction case where it is shown that a transition τ denoted by a CafeOBJ action operator *a* preserves *p*, we suppose that we need to perform case analysis based on two predicates *q* and *r* on Υ. For example, the case is split into the four sub-cases: (1) $q \wedge r$, (2) $q \wedge \neg r$, (3) $\neg q \wedge r$ and (4) $\neg q \wedge \neg r$. We should write a (proof) passage of proof scores for each case. We show a proof passage for case (2) in Fig. 1. A comment starts with `--` and terminates at the end of the line. The constant s denotes an arbitrary state of $\mathcal{S}$. The first two equations say that *q* holds but r does not in the state denoted by s. The third equation denotes the induction hypothesis. Then the last line checks if τ denoted by *a* preserves *p*. In our way of writing proof scores used so far, case analysis should be done entirely by hand and the proof passage for each case obtained by the case analysis should be written by hand, which can lead to human errors such that some cases to check may be overlooked.

In this section, we describe the two methods of mechanically supporting case analysis.

A. Representing Cases with Operators

Atomic formulas are terms whose sorts are `Bool` and that do not include any logical operators. A literal is an atomic formula (a positive literal) or the negation of an atomic formula (a negative literal). A literal may be called a basic predicate, which is denoted by $\hat{p}$, in this paper.

States are denoted by CafeOBJ terms, which are called *state constants*. State constants are classified into atomic and composite ones.

Definition 5 (State constant): An *atomic state constant* (ASC) is a constant that denotes an arbitrary state where a basic predicate holds. *Composite state constants* (CSCs) are inductively defined as follows:

- An ASC is a CSC.
- If s_1 and s_2 are CSCs, then $s_1 \otimes s_2$ is a CSC.

Only an object that is thus constructed is a CSC. □

A CSC that denotes an arbitrary state where a predicate p holds is denoted by s_p. A CSC $s_p \otimes s_q$ denotes an arbitrary state where both predicates p and q hold. The CSC is also denoted by $s_{p \wedge q}$. For example, let us consider three basic predicates $\hat{p}$, $\hat{q}$ and $\hat{r}$. Six ASCs $s_{\hat{p}}$, $s_{\neg\hat{p}}$, $s_{\hat{q}}$, $s_{\neg\hat{q}}$, $s_{\hat{r}}$ and $s_{\neg\hat{r}}$ are declared to construct the eight CSCs obtained based on the three basic predicates. For example, $s_{\hat{p}} \otimes s_{\hat{q}} \otimes s_{\hat{q}}$ denotes an arbitrary state where $\hat{p} \wedge \hat{q} \wedge \hat{r}$ holds and $s_{\neg\hat{p}} \otimes s_{\hat{q}} \otimes s_{\neg\hat{q}}$ denotes an arbitrary state where $\neg\hat{p} \wedge \hat{q} \wedge \neg\hat{r}$ holds.

CSCs that denote an arbitrary state where a predicate p holds is constructed as follows:

- p is transformed into a logically equivalent disjunctive normal form $(\hat{p}^1_1 \wedge \ldots \wedge \hat{p}^1_{m_1}) \vee \ldots \vee (\hat{p}^n_1 \wedge \ldots \wedge \hat{p}^n_{m_n})$ where each $\hat{p}^j_i$ is a basic predicate.
- Each disjunct $\hat{p}^j_1 \wedge \ldots \wedge \hat{p}^j_{m_j}$ is denoted by the CSC $s_{\hat{p}^j_1} \otimes \ldots \otimes s_{\hat{p}^j_{m_j}}$ that is obtained by replacing each $\hat{p}^j_i$ and $\wedge$ with the ASC $s_{\hat{p}^j_i}$ and $\otimes$, respectively.

The n CSCs obtained through this procedure denote an arbitrary state where p holds.

When a predicate p is transformed into a logically equivalent disjunctive normal form, some disjuncts of the disjunctive normal form may include contradictions such as $\hat{p}^j_i \wedge \neg\hat{p}^j_i$. Since there are no states denoted by such disjuncts, the disjuncts can be deleted from the disjunctive normal form. Although contradictions in some disjuncts of the disjunctive normal form do not affect the soundness of verification at all, we can save time and space taken by verification by deleting the disjuncts from the disjunctive normal form. It is extremely difficult to find all disjuncts that include contradictions. Instead of finding all conjuncts including contradictions, we use a simple but useful method. For each ASC $s_{\hat{p}^j_i}$ used to denote the predicate p, the list $tbl(s_{\hat{p}^j_i})$ of ASCs that contradict $s_{\hat{p}^j_i}$ is prepared. The list $tbl(s_{\hat{p}^j_i})$ surely includes $s_{\neg\hat{p}^j_i}$. Such lists are constructed by hand, but given such lists, finding disjuncts that include contradictions is automated.

B. Exhausting All Cases Using Matrices

In this subsection, we describe the first method of mechanically supporting case analysis. We still suppose that we prove p invariant to $\mathcal{S}$ by induction on the number of transitions applied. Since the base case is often straightforward to prove, we focus on each induction case where we prove that each transition preserves p.

The basic idea of the method proposed here is to use a matrix to represent all possible sub-cases obtained by case analysis to prove p invariant to $\mathcal{S}$. Each element of such a matrix is a CSC that denotes an arbitrary state where some predicate holds. Besides, all the elements of the matrix cover all the necessary sub-cases to consider.

For each pair (τ, p) of transitions of $\mathcal{S}$ and predicates to be proved invariant to $\mathcal{S}$, one matrix is constructed. We first construct CSCs that denote an arbitrary state where the effective condition c_τ of τ holds. Let the CSCs be $c^\tau_1, \ldots, c^\tau_m$, and C be the list of the CSCs. We next construct CSCs that denote an arbitrary state where p holds. Let the CSCs be $p_1, \ldots, p_n$, and P be the list of the CSCs. The matrix for the pair (τ, p) is made of C and P, namely $C \times P$. The matrix is shown in Fig. 2. Such matrices are called *CA-matrices*. For each element $c^\tau_i(s) \otimes p_j(s)$, $c^\tau_i(s)$ denotes a sub-case to consider, where c_τ holds, and $p_j(s)$ denotes an induction

$$\begin{array}{c|ccc}
 & p_1(s) & \ldots & p_n(s) \\
\hline
c_1^{\tau}(s) & c_1^{\tau}(s) \otimes p_1(s) & \ldots & c_1^{\tau}(s) \otimes p_n(s) \\
\vdots & \vdots & \ddots & \vdots \\
c_m^{\tau}(s) & c_m^{\tau}(s) \otimes p_1(s) & \ldots & c_m^{\tau}(s) \otimes p_n(s)
\end{array}$$

Fig. 2. A matrix for the pair (τ, p).

hypothesis. The CA-matrix $C \times P$ covers all the necessary cases for the induction case showing that τ preserves p. The reason is as follows:

- From Definition 1, τ does not change anything if it is applied in a state where c_τ does not hold. Consequently, τ surely preserves p in an arbitrary state where c_τ does not hold, which means that we do not have to consider such a state.
- The induction used guarantees that we only have to consider an arbitrary state s where p holds; we can use the induction hypothesis saying that p holds in s.
- From the above two facts, we only need to consider an arbitrary state where both c_τ and p hold. The CSCs $c_1^{\tau}, \ldots, c_m^{\tau}$ cover all the cases where c_τ holds by construction and the CSCs $p_1, \ldots, p_n$ cover all the cases where p holds by construction. Therefore, all the elements of the CA-matrix $C \times P$ overs all the necessary cases for the induction case showing that τ preserves p.

For each element $c_i^{\tau}(s) \otimes p_j(s)$, $p(\tau(c_i^{\tau}(s) \otimes p_j(s)))$ is reduced. If the result is as expected, namely true, it is shown that τ preserves p in the sub-case denoted by $c_i^{\tau}(s) \otimes p_j(s)$. If the result is false, there are two possibilities that an arbitrary state corresponding to $c_i^{\tau}(s) \otimes p_j(s)$ is unreachable and there is a counterexample showing that p is not invariant to $\mathcal{S}$, respectively. The first possibility is usually much more than the second one from our experiences. We only examine the first possibility unless we can easily find such a counterexample. For the first possibility, we should find a lemma and prove the lemma invariant to $\mathcal{S}$ likewise. It is often the case that two ASCs in $c_i^{\tau}(s) \otimes p_j(s)$ contradict each other. Therefore, such a lemmas is most likely the negation of the predicate denoted by two such ASCs. If the result is neither true nor false, we should split the sub-case denoted by $c_i^{\tau}(s) \otimes p_j(s)$ furthermore, which is described in the coming subsection.

CA-matrices are straightforwardly implemented in CafeOBJ as terms, and reducing $p(\tau(c_i^{\tau}(s) \otimes p_j(s)))$ and checking if the result is true can be automated by the CafeOBJ system.

Case analysis with CA-matrices has two advantages.

1) All necessary sub-cases to consider are surely covered, which prevents users from overlooking some sub-cases.
2) CA-matrices constructed for a predicate to be proved invariant to $\mathcal{S}$ can be reused for other predicates to be proved.

Since the first advantage has been described in detail, we describe the second advantage in detail. In order to prove a predicate invariant to $\mathcal{S}$, we often need to prove other predicates invariant to $\mathcal{S}$ as lemmas. We thus usually prove multiple predicates invariant to $\mathcal{S}$. On the other hand, transitions are fixed to $\mathcal{S}$. Therefore, we make a predicate part a parameter when constructing CA-matrices for the transitions of $\mathcal{S}$ and construct templates of CA-matrices. Given a concrete predicate p, the templates of CA-matrices are instantiated to construct the CA-matrices for the pair (τ, p) for each transition τ of $\mathcal{S}$.

For convenience, we actually make one CA-matrix to prove a predicate p invariant to $\mathcal{S}$ by concatenating multiple CA-matrices each of which is constructed for the pair (τ, p) for each transition τ of $\mathcal{S}$.

C. More Precise Case Analysis

As described in the previous subsection, for the result obtained by reducing $p(\tau(c_i^{\tau}(s) \otimes p_j(s)))$ for some element $c_i^{\tau}(s) \otimes p_j(s)$ of the CA-matrix for the pair (τ, p) to be neither true nor false indicates that the sub-case corresponding to the element should be split furthermore. In order to split a sub-case more precisely, we find a set of basic predicates from the specification of $\mathcal{S}$. Based on the set of basic predicates, we construct a list $A = \{(s_{\hat{q}}, s_{\neg\hat{q}}), \ldots\}$ of complementary pairs of ASCs. Each pair $(s_{\hat{q}}, s_{\neg\hat{q}})$ is a candidate that can be used for more precise case analysis. We describe a method of mechanically selecting the most likely useful candidates from A.

Let $c_i^{\tau}(s) \otimes p_j(s)$ be an element of the CA-matrix for the pair (τ, p) such that the result obtained by reducing $p(\tau(c_i^{\tau}(s) \otimes p_j(s)))$ is neither true nor false. For each pair $(s_{\hat{q}}, s_{\neg\hat{q}}) \in A$, we reduce the two terms $p(\tau(c_i^{\tau}(s) \otimes p_j(s) \otimes s_{\hat{q}}))$ and $p(\tau(c_i^{\tau}(s) \otimes p_j(s) \otimes s_{\neg\hat{q}}))$. Based on the results obtained by reducing the two terms, we decide what to do next as follows:

1) If both results are true, then we have successfully finished showing that τ preserves p in an arbitrary state where the predicate denoted by $c_i^{\tau}(s) \otimes p_j(s)$ holds.
2) If one of the results is true and the other is false, then the sub-case where the result is false deserves to be considered. Let s_f be the CSC corresponding to the sub-case. An arbitrary state corresponding to s_f may be unreachable. In order to show this, we should find a lemma and prove the lemma invariant to $\mathcal{S}$ in the same way of proving p invariant to $\mathcal{S}$. It is often the case that two ASCs in s_f contradict each other. Therefore, such a lemma is most likely the negation of the predicate denoted by two such ASCs.
   Perhaps s_f may indicate a counterexample showing that p is not invariant to $\mathcal{S}$. But this possibility is usually much less than the first one from our experiences.
3) If one of the results is true and the other is neither true nor false, then the sub-case where the result is neither true nor false should be split furthermore using A.
4) If both results are false, we do the same thing as we do for the sub-case denoted by s_f.
5) If both results are neither true nor false, the both sub-cases are split furthermore using A.

We first try to find a candidate that belongs to category 1) from A. If no such a candidate is found, we next try to find a candidate that belongs to category 2) from A. If no such a candidate is found, we repeatedly do the same thing until we pick up an arbitrary candidate that belongs to category 5). From our experiences, it is very often the case that we can find a candidate that belongs to category 1), 2) or 3) from A.

The two ASCs $s_{\hat{q}}$ and $s_{\neg\hat{q}}$ surely cover the whole cases because $s_{\hat{q}}$ corresponds to an arbitrary state where $\hat{q}$ holds and $s_{\neg\hat{q}}$ to an arbitrary state where $\hat{q}$ does not. Therefore, the case splitting based on a complementary pair $(s_{\hat{q}}, s_{\neg\hat{q}})$ of ASCs does not break the desired property that CA-matrices have, i.e. CA-matrices cover all necessary cases for induction cases showing that transitions preserve predicates.

Given A, which is implemented in CafeOBJ as a CafeOBJ term, what was described in this subsection is mostly automated by the CafeOBJ system.

D. How to Use Lemmas

We suppose that s_f is a CSC such that the result of reducing $p(\tau(s_f))$ is false in the proof that p is invariant to $\mathcal{S}$, and that a predicate q, which is supposed to have been proved invariant to $\mathcal{S}$, is used as a lemma for the case corresponding to the CSC s_f. We construct CSCs that denote an arbitrary state where q holds. Let the CSCs be $q_1, \ldots, q_l$. Instead of $p(\tau(s_f))$, we reduce the l terms $p(\tau(s_f \otimes q_1)), \ldots, p(\tau(s_f \otimes q_l))$ because since q is invariant to $\mathcal{S}$, we only need to consider an arbitrary state where q holds and the l CSCs cover all cases such that q holds by construction. If all the results are true, we can conclude that τ preserves p in an arbitrary state corresponding to the CSC s_f, or such a state is not reachable with respect to $\mathcal{S}$.

E. How to Deal with Non-state Basic Predicates

Basic predicates that can be well dealt with by the two methods described in the previous two subsections are state basic predicates, which take a state as one of their parameters. But, non-state basic predicates are also used in specifications and predicates to prove. For example, the equivalence predicate _=_ for each data type is a non-state basic predicate. Case analysis based on non-state basic predicates should be done manually. After the first method is used, manual case analysis based on non-state basic predicates are done.

F. CafeOBJ Representation of ASCs and CSCs

We describe how to represent ASCs and CSCs as CafeOBJ terms.

CafeOBJ is order-sorted, meaning that partial order among sorts can be defined. By declaring $S_2 < S_1$, every term whose sort is S_2 is also a term whose sort is S_1. ASCs are denoted by a hidden sort, say A, and CSCs are denoted by a hidden sort, say C. Besides, C is declared as a subsort of the hidden sort H denoting the state space Υ and A is declared as a subsort of C. A and C, together with H, are declared as follows:

[A < C < H]

The composition operator $\otimes$ is denoted by the CafeOBJ operator declared as follows:

op _o_ : C C -> C {assoc comm coherent}

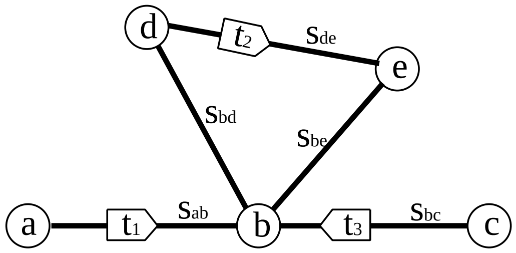

Fig. 3. An example of single-track railroads.

assoc, comm and coherent are attributes given to the operator _o_. assoc declares that the operator is associative (namely $(c_1\ o\ c_2)\ o\ c_3 = c_1\ o\ (c_2\ o\ c_3)$), comm declares that the operator is commutative (namely $c_1\ o\ c_2 = c_2\ o\ c_1$, and coherent declares that the operator is neither observation nor action operators.

Let $\hat{p}$ be the CafeOBJ operator that denotes a basic predicate $\hat{p}$, and $s_{\hat{p}}$ and $s_{\neg\hat{p}}$ be the CafeOBJ constants that denote the two ASCs $s_{\hat{p}}$ and $s_{\neg\hat{p}}$ for the basic predicate $\hat{p}$. Basically the two ASCs are then defined with the following two equations, respectively:

ceq $\hat{p}(s_{\hat{p}}\ o\ C_1) =$ true if $comp(s_{\hat{p}}, C_1)$.
ceq $\hat{p}(s_{\neg\hat{p}}\ o\ C_1) =$ false if $comp(s_{\neg\hat{p}}, C_1)$.

C_1 is a CafeOBJ variable whose sort is C. *comp* is the CafeOBJ operator that returns true if an ASC $s_{\hat{p}}$ given as its first argument and a CSC C_1 given as its second argument are composable. The operator *comp* searches the lits $tbl(s_{\hat{p}})$ (see Subsect. III-A) to check if $s_{\hat{p}}$ and C_1 are composable. If an ASC in C_1 is found in the list, then $s_{\hat{p}}$ and C_1 are not composable. Otherwise, they are composable.

IV. A Case Study

We describe formal verification that no collisions occur in any single-track railroad that adopts the staff system, which is a railroad signaling system, with the proposed methods.

A. The Staff System

The staff system (or the tablet blocking system) is a railroad signaling system for single-track railroads. It prevents trains from colliding on the railroad track between arbitrary two adjacent stations.

We model single-track railroads as undirected graphs. Figure 3 shows an undirected graph that represents a single-track railroad. Nodes represent stations. There are five stations in the single-track railroad shown in Fig. 3, which are represented by the five nodes a, b, c, d and e. Edges represent railroad tracks. There are five railroad tracks in the single-track railroad, which are represented by the five edges e_{ab}, e_{bc}, e_{bd}, e_{be} and e_{de}. We suppose that there exists at most one edge between any two nodes x and y, and the edge is denoted by e_{xy} if any. We also suppose that any two edges do not cross at grade.

In any single-track railroad that we are going to consider, there are the same number of tokens as that of edges, each token exactly corresponds to one of the edges and vice versa, and there are an arbitrary number of trains. If a train has a token, the train is allowed to enter the edge corresponding to the token. In the single-track railroad shown in Fig. 3, there are three trains that are denoted by t_1, t_2 and t_3, which are to have the tokens corresponding to e_{ab}, e_{bc} and e_{de}, respectively. Initially, each train stops at a node and does not have any tokens, and each token is owned by one of the nodes connected to the edge corresponding to the token.

Trains move according to the following rules:

- If a train is at a node and has the token corresponding to an edge connected to the node, then the train can move to the edge.
- If a train is at an edge, then the train can move to one of the nodes connected to the edge.
- If a train is at a node and the node has a token, then the train can obtain the token.
- If a train is at a node and has a token, then the train can return the token to the node.

B. Modeling and Specification

We model an arbitrary single-track railroad that adopts the staff system as an OTS, and the OTS is specified in CafeOBJ.

The two parameterized observes pos_{tr} and staff_{ed} are used, which are defined as follows:

- pos_{tr} takes a state s and returns the ID of the place (either a station or a railroad track) where the train denoted by a train ID *tr* is in the state s. Note that stations are denoted by nodes and railroad tracks are denoted by edges in undirected graphs modeling single-track railroads.
- staff_{ed} takes a state s and returns the ID of the current owner (either a station or a train) of the token corresponding to the edge denoted by an edge ID *ed* in the state s.

The two parameterized observers are denoted by the two CafeOBJ observation operators declared as follows:

```
bop pos   : TrID   State -> TcID
bop staff : EdgeID State -> StaffPos
```

`State` is the hidden sort denoting the state space. `TrID` and `EdgeID` are the visible sorts denoting train IDs and edge IDs, respectively. `TcID` is a super sort of the two visible sorts `EdgeID` and `NodeID`, where `NodeID` denotes node IDs, which means that `TcID` denotes both edge IDs and node IDs. `StaffPos` is a super sort of the two visible sorts `TrID` and `NodeID`.

Let `init` be the constant denoting an arbitrary initial state. The initial conditions satisfied by the single-track railroad is defined as follows:

```
eq isNode(pos(TR,init))            = true .
eq isNode(staff(ED,init))          = true .
eq isAdjacent(staff(ED,init),ED) = true .
```

`TR` and `ED` are CafeOBJ variables whose sorts are `TrID` and `EdgeID`, respectively. `isNode` is the predicate that checks if a given argument is a node, and `isAdjacent` is the predicate that checks if one argument is a node, the other is an edge and they are connected. The first equation says that every train denoted by `TR` is initially at a node and the remaining two equations say that the initial owner of each token corresponding to the edge denoted by `ED` is a node connected to the edge.

The four parameterized transitions $\text{move-to-edge}_{tr,ed}$, $\text{move-to-node}_{tr,nd}$, $\text{catch}_{tr,ed,nd}$ and $\text{release}_{tr,ed,nd}$ are used, which are defined as follows:

- $\text{move-to-edge}_{tr,ed}$ takes a state s and makes the train denoted by a train ID *tr* move to the edge denoted by an edge ID *ed* in the successor state if the train is at a node connected to the edge and has the token corresponding to the edge in the state s.
- $\text{move-to-node}_{tr,nd}$ takes a state s and makes the train denoted by a train ID *tr* move to the node denoted by a node ID *nd* in the successor state if the train is at an edge connected to the node in the state s.
- $\text{catch}_{tr,ed,nd}$ takes a state s and makes the train denoted by a train ID *tr* obtain the token corresponding to the edge denoted by an edge ID *ed* from the node denoted by a node ID *nd* in the successor state if if the train is at the node and the node has the token in the state s.
- $\text{release}_{tr,ed,nd}$ takes a state s and makes the train denoted by a train ID *tr* return the token corresponding to the edge denoted by an edge ID *ed* to the node denoted by a node ID *nd* in the successor state if the train is at the node and has the token.

The four parameterized transitions exactly correspond to the four rules described in the previous subsection.

The four parameterized transitions are denoted by the four CafeOBJ action operators, whose declarations are shown in Fig. 4. Let `STAFFSYSTEM` be the module where the OTS under consideration is specified in CafeOBJ.

C. Verification

We describe the verification that no collisions occur in any single-track railroad that adopts the staff system. For the verification, it suffices to prove that the OTS under consideration has the invariant property described as follows:

$$\text{invariant } (\text{pos}_{tr1}(s) = cs \wedge \text{pos}_{tr2}(s) = cs) \Rightarrow tr1 = tr2 \quad (1)$$

The invariant says that if a train denoted by *tr*1 is at an edge denoted by *cs* and a train denoted by *tr*2 is also at the same edge, then the two trains are always the same, which means that for each edge there is always at most one train at the edge, implying that no collisions occur.

1) Representing Cases with Operators: We declare a module `CASES` that imports the module `STAFFSYSTEM`. In the module `CASES`, hidden sorts `Atom` (denoting ASCs) and `Case` (denoting CSCs) are declared as a subsort of `Case` and a subsort of `State`, respectively, and necessary operators such as `_o_`, `comp` and `tbl` and necessary constants denoting arbitrary values are declared. Besides, ASCs for basic predicates used in the predicate to prove and the effective conditions of the transitions of the OTS under consideration are declared and defined. We are about to describe these ASCs.

The predicate to be proved invariant to the OTS under consideration contains three basic predicates, which are as follows:

```
act move-to-edge : TR':TrID ED':EdgeID S:State -> State .
    cond (isAdjacent(pos(TR',S),ED') and staff(ED',S) = TR') .
    eff pos(TR:TrID,S':State) := if TR = TR' then ED' else pos(TR,S) fi .
    eff staff(ED:EdgeID,S':State) := staff(ED,S) .
tca

act move-to-node : TR':TrID ND':NodeID S:State -> State .
    cond isAdjacent(pos(TR',S),ND') .
    eff pos(TR:TrID,S':State) := if TR = TR' then ND' else pos(TR,S) fi .
    eff staff(ED:EdgeID,S':State) := staff(ED,S) .
tca

act catch : TR':TrID ED':EdgeID ND':NodeID S:State -> State .
    cond (pos(TR',S) = ND' and staff(ED',S) = ND') .
    eff pos(TR:TrID,S':State) := pos(TR,S) .
    eff staff(ED:EdgeID,S':State) := if ED = ED' then TR' else staff(ED,S) fi .
tca

act release : TR':TrID ED':EdgeID ND':NodeID S:State -> State .
    cond (pos(TR',S) = ND' and staff(ED',S) = TR') .
    eff pos(TR:TrID,S':State) := pos (TR,S) .
    eff staff(ED:EdgeID,S':State) := if ED = ED' then ND' else staff(ED,S) fi .
tca
```

Fig. 4. Specification of the four CafeOBJ action operators.

- `tr1 = tr2`
- `pos(tr1,s) = cs`
- `pos(tr2,s) = cs`

`tr1` and `tr2` are constants whose sorts re `TrID`, `cs` is a constant whose sort is EdgeID and `s` is a constant whose sort is `State`.

The first basic predicate does not include any constants (namely `s`) denoting states, and we do not use any ASCs for it. For the second basic predicate, we use the two ASCs `tr1@cs` and `~tr1@cs`, which denote an arbitrary state where the predicate holds and it does not hold, respectively. For the third basic predicate, we use the two ASCs `tr2@cs` and `~tr2@cs`, which denote an arbitrary state where the predicate holds and it does not hold, respectively.

We also make ASCs for the basic predicates appearing in the effective conditions of the transitions of the OTS under consideration. In this paper, we only show the basic predicates appearing in the effective condition of the transition move-to-edge$_{tr,ed}$. The basic predicates are as follows:

- `isAdjacent(pos(tr,s),ed)`
- `staff(ed,s) = tr`

`tr` and `ed` are constants whose sorts are `TrID` and `EdgeID`, respectively. For the first predicate, we use the two ASCs `adj` and `~adj`. For the second predicate, we use the two ASCs `sted@tr` and `~sted@tr`.

The ASCs are defined with equations. In this paper, we only show the equations that define the two ASCs `tr1@cs` and `~tr1@cs`, which are as follows:

```
ceq (pos(tr1,(tr1@cs o C)) = cs)
    = true if comp(tr1@cs,C) .
ceq (pos(tr1,(~tr1@cs o C)) = cs)
    = false if comp(~tr1@cs,C) .
```

`C` is a CafeOBJ variable whose sort is `Case`.

2) Exhausting All Cases: We declare a module `PROOF` that imports the module `CASES`. In the module `PROOF`, we declare the constants `INI` and `IND` whose sorts are `Bool`, which denote the base case and the induction cases, respectively. Besides, a template of CA-matrices is made in the module. We are about to describe the template.

For each parametrized transition, we declare an operator denoting the corresponding induction case that the transition preserves the predicate concerned. For example, for the transition move-to-edge$_{tr,ed}$, we declared the following operator:

```
op MOVE-TO-EDGE :
          TrID EdgeID Case Case -> Bool
```

The operators denoting the induction cases are defined in equations. For example, `MOVE-TO-EDGE` is defined as follows:

```
ceq MOVE-TO-EDGE (TR,ED,C,C') = true
  if comps(C,C') implies
     (p(C o C') implies
      p(move-to-edge(TR,ED,(C o C')))) .
```

`TR`, `ED`, `C` and `C'` are CafeOBJ variables whose sorts are `TrID`, `EdgeID`, `Case` and `Case`, respectively.

Given the operators denoting the induction cases, we can construct a template of CA-matrices, which is denoted by the following declared and defined operator:

```
op FORALL-ACTION : Case -> Bool
eq FORALL-ACTION(C) =
   MOVE-TO-EDGE(tr,ed,
                adj o tr@nd o sted@tr,C)
 | MOVE-TO-NODE(tr,nd,adj o tr@ed,C)
 | CATCH(tr,ed,nd,tr@nd o sted@nd,C)
 | RELEASE(tr,ed,nd,tr@nd o sted@tr,C) .
```

`C` is a CafeOBJ variable whose sort is `Case`, which is the parameter of the template of CA-matrices. The operator `_|_`

```
mod CLAIM1 { ex(PROOF)
 -- the predicate to prove
 eq p(S:State) = (tr1 = tr2) or
                 not(pos(tr1,S) = cs) or
                 not(pos(tr2,S) = cs) .
 -- assumption
 eq (tr1 = tr2) = false .
 -- base case
 eq INI = p(init) .
 -- induction hypotheses.
 ops hyp1 hyp2 : -> Case
 eq hyp1 = ~tr1@cs .
 eq hyp2 = ~tr2@cs .
 -- induction cases.
 eq IND = FORALL-ACTION(hyp1)
       | FORALL-ACTION(hyp2) .
}
```

Fig. 5. The module `CLAIM1`.

is the constructor of sets of terms whose sorts are `Bool` and basically equivalent to `_and_`.

In order to prove the predicate concerned invariant to the OTS under consideration, we declare a module `CLAIM1`, which is shown in Fig. 5. `ex(PROOF)` declares the importation of the module `PROOF`. The operator `p` denotes the predicate concerned, and `S` is a CafeOBJ variable whose sort is `State`. The term on the right-hand side of the equation defining `p` denotes an disjunctive normal form of the predicate. The two constants `hyp1` and `hyp2` denote arbitrary states (defined as `~tr1@cs` and `~tr2@cs`) where the two induction hypotheses (`not(pos(tr1,S) = cs)` and `not(pos(tr2,S) = cs)`) hold, respectively. By using the two constants, the template of CA-matrices is instantiated. One of the three disjuncts of the predicate (namely `tr1 = tr2`) does not have any states and then its truth value does not change at all as state transitions go on. Consequently, if the disjunct is used as the induction hypothesis, then every transition surely preserves the predicate. Therefore, we assume that the disjunct does not hold for the verification. The assumption is given as the equation in the module `CLAIM1`.

The term `INI | IND` denotes the proof candidate of the invariant. If the result obtained by reducing the term is true, then the term is surely the proof. Otherwise, basically we should do more case analysis and find necessary lemmas. The actual result obtained by reducing the term is as follows:

```
  MOVE-TO-EDGE(tr,ed,
               adj o tr@nd o sted@tr,~tr1@cs)
| MOVE-TO-NODE(tr,nd,adj o tr@ed,~tr1@cs)
| MOVE-TO-EDGE(tr,ed,
               adj o tr@nd o sted@tr,~tr2@cs)
| MOVE-TO-NODE(tr,nd,adj o tr@ed,~tr2@cs)
```

The result means that four out of nine cases (namely one for the base case and eight for the induction cases) have not been proved yet.

What to do next is manual case analysis based on non-state basic predicates. Based on the equivalence predicates for `TrID` and `EdgeID` plus the five constants `tr`, `tr1`, `tr2`, `cs` and `ed`, we consider the following five cases:

1) $\mathtt{tr} \neq \mathtt{tr1} \wedge \mathtt{tr} \neq \mathtt{tr2}$
2) $\mathtt{tr} = \mathtt{tr1} \wedge \mathtt{tr} \neq \mathtt{tr2} \wedge \mathtt{cs} \neq \mathtt{ed}$
3) $\mathtt{tr} = \mathtt{tr1} \wedge \mathtt{tr} \neq \mathtt{tr2} \wedge \mathtt{cs} = \mathtt{ed}$
4) $\mathtt{tr} \neq \mathtt{tr1} \wedge \mathtt{tr} = \mathtt{tr2} \wedge \mathtt{cs} \neq \mathtt{ed}$
5) $\wedge \mathtt{tr} \neq \mathtt{tr1} \wedge \mathtt{tr} = \mathtt{tr2} \wedge \mathtt{cs} = \mathtt{ed}$

For the cases 1), 2) and 4), `INI | IND` is reduced to true as expected. For the remaining cases 3) and 5), we use the second method, namely the more precise case analysis.

Let us show the proof passage of the case 3), which is as follows:

```
open CLAIM1
 eq (tr = tr1) = true .
 eq (tr = tr2) = false .
 eq (cs = ed) = true .
 red INI | IND .
close
```

The command `open` makes a temporary module that imports a module taken as the argument, and the command `close` destroys the temporary module.

For the case 3), the result is as follows:

```
MOVE-TO-EDGE(tr,ed,
             adj o tr@nd o sted@tr,~tr1@cs)
```

The result says that it is not shown yet that the transition move-to-edge$_{tr,ed}$ preserves the predicate concerned in the case characterized by the CSC `adj o tr@nd o sted@tr` plus the three equations appearing in the proof passage.

3) More Precise Case Analysis: We describe the more precise case analysis for the case 3). For the more precise case analysis, we first select the five basic predicates. The pairs of the ASCs corresponding to the five basic predicates are (`tr@nd`,`~tr@nd`), (`tr@ed`,`~tr@ed`), (`tr1@cs`, `~tr1@cs`), (`tr2@cs`,`~tr2@cs`) and (`adj`,`~adj`).

The following CafeOBJ passage is used to do the more precise case analysis for the case 3):

```
open CLAIM1
 eq (tr1 = tr2) = false .
 eq (tr = tr1) = true .
 eq (tr = tr2) = false .
 eq (cs = ed) = true .
 eq act(S:State) = move-to-edge(tr,ed,S) .
 red traverse(atomlist,
            adj o tr@nd o sted@tr o ~tr1@cs) .
close
```

The constant `atomlist` is defined as the following term:

```
< tr@nd,~tr@nd >   :: < tr@ed,~tr@ed >   ::
< tr1@cs,~tr1@cs > :: < tr2@cs,~tr2@cs > ::
< adj,~adj >       :: empty
```

The term denotes the list of the pairs of the ASCs corresponding to the five basic predicates. The operator `traverse` performs the more precise case analysis. Given a list of ASC pairs and a CSC `c`, for each ASC `a` in the list the operator `traverse` reduces the following term:

```
comp(a,c) implies
            (p(a o c) implies p(act(a o c)))
```

The result obtained by reducing the term `traverse(...)` in the CafeOBJ passage is as follows:

```
< tr@nd,tt >    :: < tr@ed,~tr@ed > ::
< tt,~tr1@cs > :: < ff,tt >        ::
< adj,tt >      :: empty
```

The constants `tt` and `ff` are ASCs corresponding to true and false, respectively.

We turn our attention to the pair `< ff,tt >` in the result, which corresponds to the pair `< tr2@cs,~tr2@cs >` in the list denoted by `atomlist`. The pair in the result suggests that an arbitrary state corresponding to the CSC `tr2@cs o adj o tr@nd o sted@tr o ~tr1@cs` plus the four equations appearing in the CafeOBJ passage may be unreachable and we may find some contradictions in the CSC plus the four equations, from which a necessary lemma may be conjectured.

Considering the equations `tr = tr1` and `cs = ed`, we know that `tr2@cs` and `sted@tr` contradict each other. Since we assume that `tr1` is different from `tr2`, we can conjecture the following lemma:

$$\mathsf{invariant}\ (tr1 = tr2 \vee \mathrm{pos}_{tr2}(s) \neq cs \vee \mathrm{staff}_{cs}(s) \neq tr1) \quad (2)$$

(2) is used to rewrite the proof passage of the case 3) as follows:

```
open CLAIM1
 eq (tr = tr1) = true .
 eq (tr = tr2) = false .
 eq (cs = ed) = true .
 eq IND = FORALL-ACTION (hyp1 o ~tr2@cs)
        | FORALL-ACTION (hyp1 o ~stcs@tr1)
        | FORALL-ACTION (hyp2) .
 eq ~stcs@tr1 = ~sted@tr .
 red INI | IND .
close
```

In the case 3), since `tr1` is different from `tr2`, we only consider the last two disjuncts in (2). The two disjuncts are used to split the case using the induction hypothesis `hyp1` into the two sub-cases. Besides, since `cs = ed` and `tr1 = tr`, `~stcs@tr1` should be equal to `~sted@tr`, which is needed to obtain the desired result. The result obtained by reducing the term `INI | IND` in the proof passage is true as expected.

We also perform the more precise case analysis for the case 5), from which in addition to (2) we know that we need another lemma, which is as follows:

$$\mathsf{invariant}\ (\mathrm{pos}_{tr1}(s) \neq cs \vee \mathrm{staff}_{cs}(s) \neq nd) \quad (3)$$

In order to prove (2) and (3), we need other lemmas. To complete the verification, we need three more lemmas, which are as follows:

$$\mathsf{invariant}\ (tr1 = tr2 \vee \mathrm{staff}_{cs}(s) \neq tr1 \vee \mathrm{staff}_{cs}(s) \neq tr2)$$
$$\mathsf{invariant}\ (\mathrm{pos}_{tr1}(s) \neq cs \vee \mathrm{pos}_{tr1}(s) \neq nd)$$
$$\mathsf{invariant}\ (\mathrm{staff}_{cs}(s) \neq nd \vee \mathrm{staff}_{cs}(s) \neq tr1)$$

The lemmas can be verified as (1).

V. Related work

Several proof assistants have been proposed. Among them are Coq[1] and Isabell/HOL[14]. They provides some automatic proof mechanisms to some extent, but basically help users construct their proofs. Users feed commands called tactics into a proof assistant to make progress on their proofs. Tactics usually reduce a proof goal into possibly multiple and hopefully simpler proof sub-goals. But, users should select appropriate tactics in order to succeed in their proofs. This means that users are required to complete their proofs on their own without any proof assistants. One of the benefits of the use of proof assistants is that proof assistants formally assure that proofs constructed are really correct.

On the other hand, the proof-score approach to verification of distributed systems, or the OTS/CafeOBJ method does not require users to have sophisticated knowledge on theorem proving in a sense that we do not have to know what deductive rules (i.e. equations) should be applied to terms denoting formulas to prove. But, the OTS/CafeOBJ method does no have any mechanisms to help users write proof scores and then writing proof scores are subject to human errors. The proposed two methods, which mechanically support case analysis, substantially reduces human errors.

Among the existing tools supporting verification of (distributed) systems with algebraic specification languages are Larch Prover (LP)[6][10] and BOBJ[8]. The design policy of LP is to make proof assistants easier-to-use especially for engineers, but users of LP are basically required to have similar skills as those needed to use other proof assistants. BOBJ is a sibling language/system of CafeOBJ. System verification with BOBJ is also another proof-score approach. Given a set of predicates, BOBJ automatically splits the case into multiple sub-cases, each of which denotes an arbitrary state where one of the predicates holds, generates the proof passage of each sub-case and check the proof passage. But, users are responsible for covering all necessary cases.

VI. Conclusion

We have proposed two methods of supporting case analysis for verifying invariant properties of distributed systems with algebraic specification languages. The first method uses CA-matrices to cover all cases. The second method, given a set of basic predicates, mostly automates splitting cases more precisely, which also can help find necessary lemmas. We have reported on a case study, showing the effectiveness of these methods.

We have also applied the two proposed methods to the verification of another railroad signaling system called the automatic block system. We have verified seven invariants related to its safety plus 36 lemmas (which are also invariants). We can verify 12 out of the 43 invariants fully automatically with CA-matrices.

The two proposed methods cannot treat non-state basic predicates because such a predicate does not have any states and it is difficult to represent a non-state basic predicate as a constant denoting a set of state where the predicate holds. Therefore, we still need some manual case analysis for such predicates, which may cause human errors. One piece of our future work is to devise a way of mechanically supporting case analysis based on non-state basic predicates.

ACKNOWLEDGMENT

The authors wish to thank anonymous referees who commented on drafts of this paper.

REFERENCES

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus Inductive Constructions*. Springer, 2004.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[3] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[4] R. Diaconescu and K. Futatsugi. Behavioural coherence in object-oriented algebraic specification. *J.UCS*, 6:74–96, 2000.

[5] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2001.

[6] S. J. Garland and J. V. Guttag. A guide to LP, the Larch prover. MIT Laboratory for Computer Science, 1991.

[7] J. A. Goguen. *Theorem Proving and Algebra*. MIT Press, (to appear).

[8] J. A. Goguen and K. Lin. Behavioral verification of distributed concurrent systems with BOBJ. In *3rd QSIC*, pages 216–235. IEEE CS Press, 2003.

[9] J. A. Goguen and G. Malcolm. A hidden agenda. *TCS*, 245:55–101, 2000.

[10] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[11] J. Hsiang and N. Dershowitz. Rewrite methods for clausal and nonclausal theorem proving. In *10th ICALP*, volume 154 of *LNCS*, pages 331–346. Springer, 1983.

[12] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[15] K. Ogata and K. Futatsugi. Modeling and verification of distributed real-time systems based on CafeOBJ. In *16th ASE*, pages 185–192. IEEE CS Press, 2001.

[16] K. Ogata and K. Futatsugi. Formal analysis of Suzuki&Kasami distributed mutual exclusion algorithm. In *5th FMOODS*, pages 181–195. Kluwer, 2002.

[17] K. Ogata and K. Futatsugi. Formal analysis of the iKP electronic payment protocols. In *1st ISSS*, volume 2609 of *LNCS*, pages 441–460. Springer, 2003.

[18] K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ method. In *6th FMOODS*, volume 2884 of *LNCS*, pages 170–184. Springer, 2003.

[19] K. Ogata, D. Yamagishi, T. Seino, and K. Futatsugi. Modeling and verification of hybrid systems based on equations. In *DIPES 2004*, pages 43–52. Kluwer, 2004.

[20] 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.

[21] T. Seino, K. Ogata, and K. Futatsugi. Supporting case analysis with algebraic specification languages. In *4th CIT*, pages 1100–1107. IEEE CS Press, 2004.

PLACE PHOTO HERE

Takahiro Seino is a postdoc researcher at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his PhD in information science from Graduate School of Information Science, JAIST in 2003. His research interests include formal methods for safety critical systems such as railroad signaling systems.

PLACE PHOTO HERE

Kazuhiro Ogata is a research expert at NEC Software Hokuriku, Ltd. He is also a visiting Associate Professor at JAIST (Japan Advanced Institute of Science and Technology). He received his PhD in engineering from Graduate School of Science and Technology, Keio University in 1995. He was research associate at JAIST from 1995 to 2001 and a researcher at SRA Key Technology Laboratory, Inc. from 2001 to 2002. Among his research interests are parallel and distributed programming languages&systems, their formal analyzes, and formal methods and tools for the analyses.

PLACE PHOTO HERE

Kokichi Futatsugi is a professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). Before getting a full professorship at JAIST in 1993, he was working for ETL (Electrotechnical Lab.) of Japanese Government and was assigned to be Chief Senior Researcher of ETL in 1992. His research interests include formal methods, software requirements&specifications, language&system design, concurrent and cooperative computing. His primary research goal is to design and develop new languages which can open up new application areas, and/or improve the current software technology. His current approach for this goal is CafeOBJ formal specification language.