Supporting Case Analysis with Algebraic Specification Languages

Takahiro Seino
Japan Advanced Institute of Science & Technology
t-seino@jaist.ac.jp

Kazuhiro Ogata
NEC Software Hokuriku, Ltd. / Japan Advanced Institute of Science & Technology
ogata@jaist.ac.jp

Kokichi Futatsugi
Japan Advanced Institute of Science & Technology
kokichi@jaist.ac.jp

Abstract

Case analysis is essential for verification of computer systems by writing proof scores in algebraic specification languages. When case analysis is performed, it is indispensable to cover all cases and find basic predicates that can be used for splitting cases. We propose two methods to support case analysis, which concern the two things. The first method uses matrices to cover all cases. The matrices consist of predicates that come from transition rules' conditions and properties to be verified. If it is not sufficient to split cases with such matrices, we must find basic predicates in the specifications of computer systems to split cases more precisely. Given a set of basic predicates, the second method mostly automates this process, which also can help find necessary lemmas. A case study in which our methods are effectively applied to a railroad signaling system is also reported.

1. Introduction

Since computer systems are pervasive and have a major impact on society, such systems must be built safely and reliably. 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. Computer systems are often modeled as transition systems. If a computer system can be modeled as a finite transition system, model checking techniques may be the most useful. Otherwise, theorem proving techniques should be used.

Many tools supporting theorem proving have been proposed such as Isabell/HOL[6] and PVS[13]. Our approach uses CafeOBJ, 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 approach is easier to learn than those based on Isabell/HOL and PVS that rely on higher-order logic.

In our approach, a computer system is modeled as an OTS (Observational Transition System), a kind of transition system, which is described in CafeOBJ. It is then verified that the systems have desired properties by writing proof scores in CafeOBJ and having the CafeOBJ system execute (or rewrite) the proof scores. We have demonstrated the effectiveness of our approach by performing several case studies[12, 10, 9, 7, 8]. However, proof scores have been entirely written by hand, which means that human errors might have occurred. To minimize human errors, writing proof scores should be mechanically supported. In this paper, we focus on case analysis.

Case analysis is essential for writing proof scores. When case analysis is performed, it is indispensable to cover all cases and find basic predicates that can be used for splitting cases. We propose two methods to support case analysis, which concern the two things. The first method uses matrices to cover all cases. The matrices consist of predicates that come from transition rules' conditions and properties to be verified. If it is not sufficient to split cases with such matrices, we must find basic predicates in the specifications of OTSs modeling computer systems to split cases more precisely. Given a set of basic predicates, the second method mostly automates this process, which also 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 a staff system (or a tablet blocking system) that is a railroad signaling system. The case study is also reported in this paper.

2 Preliminaries

2.1 CafeOBJ in a Nutshell

CafeOBJ[1, 3] can be used to specify abstract machines as well as abstract data types. A visible sort denotes an abstract data type, while a hidden sort the state space of an abstract machine. There are two kinds of operators on hidden sorts: action and observation operators. An action operator can change states of an abstract machine. Only observation operators can be used to observe the inside of an abstract machine. An action operator is basically specified with equations by describing how the value of each observation operator changes. Declarations of observation and action operators start with `bop` or `bops`, and those of other operators with `op` or `ops`. Declarations of equations start with `eq`, and those of conditional ones with `ceq`. The CafeOBJ system rewrites a given term by regarding equations as left-to-right rewrite rules.

2.2 Observational Transition Systems

We assume that there exists a universal state space called Υ. We also suppose that each data type used has been defined beforehand, including the equivalence between two data values v_1, v_2 denoted by $v_1 = v_2$. An OTS[11] (Observational Transition System) $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{O}$: A set of observable values. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \to D$, where D is a data type. Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the equivalence between two states, denoted by $v_1 =_{\mathcal{S}} v_2$, w.r.t. $\mathcal{S}$ is defined as $\forall o \in \mathcal{O}.o(v_1) = o(v_2)$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subset \Upsilon$.
- $\mathcal{T}$: A set of conditional transition rules. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon/=_{\mathcal{S}} \to \Upsilon/=_{\mathcal{S}}$ on equivalence classes of Υ w.r.t. $=_{\mathcal{S}}$. Let $\tau(v)$ be the representative element of $\tau([v])$ for each $v \in \Upsilon$ and it is called *the successor state* of v w.r.t. τ. The condition c_τ for a transition rule $\tau \in \mathcal{T}$ is called *the effective condition*. The effective condition is supposed to satisfy the following requirement: given a state $v \in \Upsilon$, if c_τ is false in v, namely τ is not *effective* in v, then $v =_{\mathcal{S}} \tau(v)$.

An OTS is described in CafeOBJ. Observable values are denoted by CafeOBJ observation operations, and transition rules by CafeOBJ action operations.

Multiple similar observable values and transition rules may be indexed. Generally, observable values and transition rules are denoted by $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, respectively, provided that $m, n \geq 0$ and we assume that there exist data types D_k such that $k \in D_k$ ($k = i_1, \ldots, i_m, j_1, \ldots, j_n$). For example, an integer array a possessed by a process p may be denoted by an observable value a_p, and the increment of the ith element of the array may be denoted by a transition rule $inc\text{-}a_{p,i}$.

An execution of $\mathcal{S}$ is an infinite sequence $v_0, v_1, \ldots$ of states satisfying:

- *Initiation*: $v_0 \in \mathcal{I}$.
- *Consecution*: For each $i \in \{0, 1, \ldots\}$, $v_{i+1} =_{\mathcal{S}} \tau(v_i)$ for some $\tau \in \mathcal{T}$.

A state is called *reachable* w.r.t. $\mathcal{S}$ iff it appears in an execution of $\mathcal{S}$. Let $\mathcal{R}_{\mathcal{S}}$ be the set of all the reachable states of $\mathcal{S}$.

All properties considered in this paper are invariants, which are defined as follows:

$$\mathsf{invariant}\ p \stackrel{\text{def}}{=} \forall v \in \mathcal{R}_{\mathcal{S}}.p(v)$$

which means that the predicate p is true in any reachable state of $\mathcal{S}$. Let $\boldsymbol{x}$ be all free variables except for one for states in p. We suppose that $\mathsf{invariant}\, p$ is interpreted as $\forall \boldsymbol{x}.(\mathsf{invariant}\, p)$ in this paper.

3. Organizing Case Analysis

3.1 Representing Cases with Operators

Given an OTS $\mathcal{S}$, we may use the term *case* to denote an arbitrary state of $\mathcal{S}$ where some condition holds. In this subsection, we will discuss how to represent cases with terms.

We first review proof scores written in case studies performed so far. We suppose that we prove that a predicate $p : \Upsilon \to \{true, false\}$ holds in any reachable state of $\mathcal{S}$ by induction on number of transition rules applied. In the inductive step for each transition rule τ, imagine that a case analysis which uses two predicates q and r on Υ is needed. Needless to say, we must consider four cases which q and r hold or not respectively. In our past case studies, we must describe a passage of proof scores like Figure 1 for each cases. In Fig. 1, the texts following `--` is a comment. The operator s denotes an arbitrary state of $\mathcal{S}$ and three equations following it declare whether each predicate holds or not in the state s. Finally, we can check whether τ preserves p in the case with `red` command. Thus, writing proof scores entirely by hand could intervene human errors, meaning that some cases might be overlooked.

In case analysis, we focus on basic predicates. A basic predicate denoted by $\hat{p}$ is a literal in which each

```
op s :  -> H .          -- s denotes an arbitrary state.
eq q(s) = true .        -- case1:  the predicate p(s) holds.
eq r(s) = true .        -- case2:  the predicate q(s) holds.
eq p(s) = true .        -- induction hypothesis.
red p(τ(s)) .
```

Figure 1. A typical passage of proof score written so far.

variable is replaced with a constant denoting an arbitrary value of the corresponding sort. A literal is a predicate of states in which at most one logical operator exists and if so, it is negation that appears outermost like $\neg\hat{p}$.

We use CafeOBJ's terms to denote states. Such terms are called *state constants*, which are classified into atomic and composite ones. A constant that denotes an arbitrary state where a basic predicate $\hat{p}$ holds is called an *atomic state constant* (will be denoted as ASC). *Composite state constants* (will be as CSC) 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.

A CSC that denote an arbitrary state where p holds is denoted by s_p. A CSC $s_p \otimes s_q$ denotes an arbitrary state where $p \wedge q$ holds, which is also denoted by $s_{p \wedge q}$.

Generally, A CSC that denotes an arbitrary state where p holds can be constructed as follows:

- p is transformed into a logically equivalent disjunctive normal form $(\hat{p}_1^1 \wedge \ldots \wedge \hat{p}_{m_1}^1) \vee \ldots \vee (\hat{p}_1^n \wedge \ldots \wedge \hat{p}_{m_n}^n)$ where each $\hat{p}_i^j$ is a literal.
- Each disjunct is denoted by a CSC that is obtained by replacing each $\hat{p}_i^j$ in the disjunct and $\wedge$ with the corresponding ASC $s_{\hat{p}_i^j}$ and $\otimes$, respectively.

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

When p is transformed into a disjunctive normal form, some conjuncts may include contradictions, meaning that are logically equivalent to false. Although this cannot cause any serious problems, such conjuncts can be omitted when the state space is split based on p because there exists no state denoted by false. We can save time and space by finding and omitting such conjuncts. It is difficult to find all such conjuncts because we have to verify that the negation of a conjunct holds in a system under consideration. Instead, we use a simple but useful method with which many such conjuncts can be found. For each ASC $s_{\hat{p}}$, we give the list $tbl(s_{\hat{p}})$ of ASCs that contradict $s_{\hat{p}}$. The list is constructed by hand, but at least the list includes $s_{\neg\hat{p}}$. Such tables are used to find conjuncts that include contradictions.

3.2 Exhausting All Cases

In this subsection, we propose a method to support case analysis. We still suppose that we prove that a predicate $p : \Upsilon \rightarrow \{true, false\}$ holds in any reachable state of an OTS by induction on number of transition rules applied. We focus on the inductive step in which we show that a transition rule τ preserves the property. The basic idea of this method is to represent all cases with a matrix. Each element of the matrix is a CSC that denotes a case where some predicate holds, and all the elements cover all the necessary cases that should be considered.

For each pair (τ, p) of transition rules and predicates to be proved, we construct one matrix. First an arbitrary state where the effective condition of τ is denoted by CSCs. Let the CSCs be $c_1^\tau, \ldots, c_m^\tau$, and C be the list of the CSCs. Next an arbitrary state where property holds is denoted by CSCs. Let the CSCs be $p_1(s), \ldots, p_n(s)$, and P be the list of the CSCs. Then the matrix for the pair (τ, p) is obtained by $C \times P$, which is shown in Fig. 2. For each element c of the matrix for the pair (τ, p), $p(\tau(c))$ is reduced. If all the results are as expected, namely `true`, we have shown that τ preserves p. Since matrices can be straightforwardly encoded as trees, or terms, reducing $p(\tau(c))$ can be automated by CafeOBJ.

Case analysis with matrices has two advantages. You are convinced that matrices cover all the necessary cases that should be considered, and matrices used for a predicate to be verified can be used for other predicates to be verified. When matrices are encoded as CafeOBJ terms, we parameterize CSCs corresponding to predicates to be verified. Since CSCs corresponding to transition rules' conditions can be used for each predicate to be verified, CafeOBJ terms encoding ma-

trices can be used for all predicates to be verified.

Additionally, since matrices for each tau can be concatenated each other, we can get one matrix for case analysis. As we will see in section 4, we can use a concatenated matrix for convenience in the actual proofs.

3.3 More Precise Case Analysis

If the results is not as expected when $p(\tau(c))$ is reduced for some element c of the matrix for the pair (τ, p), then the cases corresponding to the element may have to be split furthermore, may need lemmas, or may indicate counter examples. From our lessons learned through several case studies, more precise case analysis should be the first choice.

We find a set of basic predicates from the specification of an OTS under consideration, from which we can make a set of complementary pairs of ASCs, say $A = \{(s_{\hat{p}}, s_{\neg\hat{p}}), \ldots\}$. Each pair $(s_{\hat{p}}, s_{\neg\hat{p}})$ is a candidate that can be used for more precise case analysis. We propose a method to find that each candidate is most likely useful.

Let c be an element of the matrix for the pair (τ, p) such that the result of reducing $p(\tau(c))$ is not as expected. We reduce the two terms $p(\tau(c \otimes s_{\hat{p}}))$ and $p(\tau(c \otimes s_{\neg\hat{p}}))$ for each pair $(s_{\hat{p}}, s_{\neg\hat{p}}) \in A$. From the result of the reduction, we can judge as follows:

1. If the result for a pair $(s_{\hat{p}}, s_{\neg\hat{p}})$ is (true, true), then we have finished showing that τ preserves p in an arbitrary state corresponding to c splitting into two case $c \otimes s_{\hat{p}}$ and $c \otimes s_{\neg\hat{p}}$.

2. If one of the results is false, then the case where the result is false deserves to be considered. Let the case and an arbitrary state where the effective condition of τ holds denote s_f and s_{c_τ} respectively. The case $s_f \otimes s_{c_\tau}$ may be a counter example or need lemmas. In our past experiences, most missing lemmas are either $s_f \otimes s_{c_\tau}$ has contradiction between the basic predicates in it, or an arbitrary state where represented $s_f \otimes s_{c_\tau}$ is unreachable.

3. The results of any pair are strange term (means that the term is not true or false), then the case where the result is so deserves to be considered. The case should be split.

We first find a candidate that belongs to (1). When there is no candidate, we focus a case which the result is false, because such case have enough information to decide that p is false. Then we find a counter example or missing lemmas from the result of (2). This process is mostly automated by CafeOBJ.

3.4 Describing Cases in CafeOBJ

The above techniques can be described in CafeOBJ.

A set of CSCs is declared subsort C of the hidden sort H representing the state space Υ of an OTS $\mathcal{S}$, and a set of ASCs is declared subsort A of C as follows:

```
*[ A < C < H ]*
```

Next, we introduce the composite operator $\otimes$ as follows:

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

where each _ represents position of each argument of the operator (in this case, `o` is declared as an infix operator). In addition, `assoc`, `comm`, and `coherent` are operator's attributes which mean it is associative, commutative, and coherent[1] respectively.

ASCs $s_{\hat{p}}$ and $s_{\neg\hat{p}}$ for a basic predicate $\hat{p}$ can be described as follows:

```
ops s_p̂ s_¬p̂ : -> A
ceq p̂(s_p̂ o C:C) = true if comp(s_p̂, C) .
ceq p̂(s_¬p̂ o C:C) = false if comp(s_¬p̂, C) .
```

where the operator `comp` returns true if an ASC given in its first argument s and a CSC `C` is composable. `comp` can be described with searching table for s.

4. Case Study

4.1 Staff System

A staff system (or tablet blocking system) is a railroad signaling system for a single-track railroad. It prevents trains from colliding on tracks connected by two adjacent stations.

We represent railroads by undirected graphs. Figure 3 shows an undirected graph that represents a railroad. A node represents a station. There are five stations in the railroad shown in Fig. 3 that are denoted by the five nodes a, b, c, d and e. An edge represents a track. There are five tracks in the railroad that are denoted 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 two nodes x and y, and if exists, the edge is denoted by e_{xy}. We also suppose that any two edges do not cross at grade.

In any 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. In the railroad shown in Fig. 3, there are three trains that are denoted by t_1, t_2 and t_3. If a train obtains a token, it means that the train is allowed to enter the edge corresponding to the token. Initially, each train stops

[1]The CafeOBJ system have special some cares for operators appearing hidden sorts in its arguments or type of returned value. In this paper, the attribute `coherent` is used to avoid these care.

	$p_1(s)$	$\dots$	$p_n(s)$
$c_1^\tau(s)$	$c_1^\tau(s) \wedge p_1(s)$	$\dots$	$c_1^\tau(s) \wedge p_n(s)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$c_m^\tau(s)$	$c_m^\tau(s) \wedge p_1(s)$	$\dots$	$c_m^\tau(s) \wedge p_n(s)$

Figure 2. A matrix for the pair (τ, p).

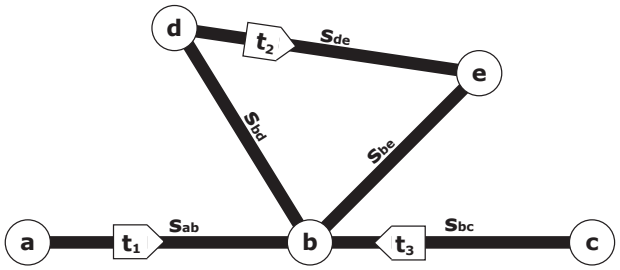

Figure 3. Example geometry of railroad with staff system

at a node and does not have any tokens, and each token is owned by one of the nodes connected by the edge corresponding to the token.

Trains work according to the following rules:

- If a train is at a node and has the token corresponding to an edge connecting 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 by 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 the token corresponding to an edge connecting the node, then the train can return the token to the node.
- If a train is at a node and has a token, then the train can return the token to the node.

We formally verify that no collisions occur in any railroad that adopts the staff system.

4.2 Description with CafeOBJ

We have modeled an arbitrary single-track railroad that adopts the staff system as an OTS, and the OTS has been described in CafeOBJ.

We have two observations `pos` and `staff`. `pos` takes a train ID (denoted by visible sort `TrID`), and returns the ID of the place at which the train is. Such a place is either a node (denoted by visible sort `NodeID`) or an edge (denoted by visible sort `EdgeID`). `staff` takes an edge ID, and returns the ID of the owner (denoted by visible sort `StaffPos`) of the token corresponding to the edge. Such a owner is either a train or a node.

The constant `init` denotes any initial state, which is defined by two equations which declare each train and token are on any node.

We also have four actions `move-to-edge`, `move-to-node`, `catch` and `release`, which model trains' moving to edges, moving to nodes, obtaining tokens and releasing tokens, respectively. The four actions exactly correspond to the four rules described in the previous subsection.

In the specification, operators `isNode` and `isAdjacent` are also declared. `isNode` is a predicate iff the place ID which is either a node or an edge ID, given as the argument is a node ID. Operator `isAdjacent` takes two place ID and returns true if one of them is a node, another one is edge, and they are connected each other in their geometry.

4.3 Verification with CafeOBJ

We describe the verification that no collisions occur in any railroad that adopts the staff system, which has been just modeled as an OTS described in CafeOBJ. The property can be expressed in CafeOBJ as follows:

Claim. invariant

```
not(tr1 = tr2) implies
    (pos(tr1,s) = cs and pos(tr2,s) = cs)
```

where constants `tr1` and `tr2` represent arbitrary elements of `TrID` respectively, and constant `cs` represents an arbitrary one of `EdgeID`. This claim says that arbitrary two trains `tr1` and `tr2` never are not in same edge `cs` simultaneously.

4.3.1 Representing Cases with Operators

To prove the claim, we will make a matrix for exhaustive case analysis. First of all, the claim tells us that we should discuss the positions of two arbitrary trains; this means that we should prepare two constants `tr1` and `tr2` on `TrID`. The claim also tells three basic predicates as follows:

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

Since the first one takes no state space, we can not prepare any ASC representing it. For the others, we can prepare four ASCs `tr1@cs`, `~tr1@cs`, `tr2@cs` and `~tr2@cs` respectively. The constant `tr1@cs` (or `~tr1@cs`) represents an arbitrary state where the predicate `pos(tr1,s) = cs` is held (or not held) respectively. `tr2@cs` and `~tr2@cs` mean likewise.

Next, we prepare ASCs for other basic predicates appearing the effective conditions of each transition rule $\tau \in \mathcal{T}$ likewise. Here, we give an example for the transition rule `move-to-edge`. Since it takes `TrID` and EdgeID, two constants `tr` and `ed` representing arbitrary elements in `TrID` and `EdgeID` respectively are prepared. And the effective condition of it tells two basic predicates as follows:

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

Therefore, we can prepare for ASCs `adj`, `~adj`, `sted@tr` and `~sted@tr` from these predicates.

4.3.2 Exhausting All Cases

Now we can create a matrix for verifying the claim. Figure 4 shows the proof module `CLAIM` for the claim. The parameterized matrix is defined another (reusable) module `PROOF`, and it is imported. The claim in DNF form, its base case and its inductive step are represented as `p`, `INI`, and `IND` respectively. CSCs `hyp2` and `hyp3` represent induction hypothesis which are corresponding to the second and third clause of the claim in DNF form.

Next, we describe some proof score for the case analysis which cannot be treated with the proposed technique. Figure 5 shows a passage of such proof score and its executed result. What is meant by this result is that the term

```
p(move-to-edge(tr,ed,
    adj o tr@nd o sted@tr o ~tr2@cs))
```

was not reduced to `true`, namely, in an arbitrary state s, where the train `tr` is at the section `nd`, the staff `ed` is owned by `tr`, and the train `tr2` is not at the section `cs`, the transition `move-to-edge(tr,ed,`s`)` does not preserve the property `p`.

Finally, we can verify the claim introducing a lemma which is discovered as will be shown in the next subsubsection. We note that the lemma is automatically verified reusing the same parameterized matrix.

4.3.3 More Precise Case Analysis

As described above, we encountered a trouble case where a term denoting the claim is not reduced to `true`. Therefore, we apply the more precise case analysis technique to the case. The upper part of Fig. 6 shows the CafeOBJ program for this analysis. The first four equations come from the passage shown in Fig. 5 and the fifth equation specifies the transition rule used in this analysis. The operator `traverse` performs this analysis with `atomlist` that is a list of pairs of known ASCs. The lower part of Fig. 6 shows the result of this analysis. In the result, `tt`, `ff` and any other term mean that the term denoting the claim in the corresponding case has been reduced to `true`, `false` and any other term. In this result, `tr2@cs` of `<<tr2@cs, ~tr2@cs>>` is replaced with `ff`, which means that the term denoting the claim has been reduced to `false` in the case characterized by `adj o tr@nd o sted@tr o hyp2 o ~tr2@cs`. We can conjecture that an arbitrary state denoted by the case is unreachable with the domain knowledge. Then, we make the lemma that an arbitrary state denoted by the case is unreachable, and prove the lemma with the first method, which can resue the module where the template of matrixes is specified. For this proof, a proof module, which is similar to the module shown in Fig. 4, is written and checked with the CafeOBJ system.

```
mod CLAIM1 { ex (PROOF)
  -- invariant.
  eq  p(S:State) = (tr1 = tr2) or-else (not pos(tr1, S) = cs) or-else (not pos(tr2, S) = cs) .
  -- base case.
  eq  INI = p(init) .
  -- cases for induction hypothesis.
  ops hyp2 hyp3 : -> Case
  eq  hyp2 = ~tr1@cs .
  eq  hyp3 = ~tr2@cs .
  -- induction step.
  eq  IND = FORALL-ACTION(hyp2) | FORALL-ACTION(hyp3) .
}
```

Figure 4. Proof module for Claim 1

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

-- reduce in %CLAIM1 : INI | IND
MOVE-TO-EDGE(tr,ed,adj o tr@nd o sted@tr,~tr2@cs) | true : Bool
(0.000 sec for parse, 657 rewrites(0.040 sec), 3001 matches)
```

Figure 5. A passage of proof score and its result for Claim 1

```
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 hyp2) .
close

-- reduce in %CLAIM1 : traverse(atomlist,adj o tr@nd o sted@tr o hyp2)
<< tr@nd, tt >> :: << tr@ed, ~tr@ed >>
  :: << tt, ~tr1@cs >> :: << ff, tt >> :: << adj, tt >> :: empty : PairList
(0.000 sec for parse, 985 rewrites(0.110 sec), 3995 matches)
```

Note that `atomlist` is given as following:

```
  eq atomlist = << tr@nd, ~tr@nd >> :: << tr@ed, ~tr@ed >>
      :: << tr1@cs, ~tr1@cs >> :: << tr2@cs, ~tr2@cs >> :: << adj, ~adj >> :: empty .
```

Figure 6. The result for inspecting the missing lemma.

5. Related work

Several proof assistants have been proposed. Among them are Isabell/HOL[6] and PVS[13] . They may automate more proof activities than the CafeOBJ system. But, they are based on higher-order logic, which is one reason why usual engineers feel that such proof assistants are difficult to use. On the other hand, the CafeOBJ system is based on rewriting, which is an efficient way of implementing equational reasoning. Equations are the most basic logical formulas and equational reasoning is the most fundamental way of reasoning, which can moderate the difficulties of proofs that might otherwise become too hard to understand. Therefore, the CafeOBJ system is superior to Isabelle/HOL in respect of easily understanding specifications and proofs. The proposed methods help automate proof activities with the CafeOBJ system.

One of the prior tools supporting verification of computer systems with algebraic specification languages is the Kumo proof assistant[5] that has been designed and implemented in the Tatami project[2] directed by Goguen. Commands fed into Kumo are written in the Duck language. Duck includes commands that implement proof rules for hidden first order logic. The proof rules transform proof tasks into subtasks. Kumo not only supports verification but also generates documents that make proofs easier to read and understand. It is also important to make proofs easier to read and understand. We think that the method of covering all cases with matrices improve understandability of proofs written in algebraic specification languages. But, the Tatami project could help further improve understandability of proofs with CafeOBJ.

Win, et al.[14] propose a methodology for proving properties of distributed systems. In the methodology, the Daikon invariant detector[4] is used, which conjecture properties that are most likely invariants in distributed systems by analyzing finite sequences of execution. Such a tool is very useful in verification with the CafeOBJ system, although conjectures might be incorrect.

6. Conclusion

We have proposed two methods of supporting case analysis for verifying invariant properties of computer systems in algebraic specification languages. The first method uses 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 a case study, showing the effectiveness of these methods.

References

[1] CafeOBJ web site. http://www.ldl.jaist.ac.jp/cafeobj/.

[2] Tatami homepage. http://www-cse.ucsd.edu/groups/tatami/.

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

[4] M. Ernst, Cokrell, W. J. Grisworld, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), 2001.

[5] J. Goguen and K. Lin. Web based support for cooperative software engineering. In J. Tsai and P. Chuang, editors, *International Symposium on Multimedia Software Engineering.*, pages 25–32. IEEE Press, 2000.

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

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

[8] K. Ogata and K. Futatsugi. Specifying and verifying a railroad crossing with CafeOBJ. In *6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (Proc. of IPDPS'2001, IEEE Computer Sciety)*, 2001.

[9] K. Ogata and K. Futatsugi. Formal analysis of suzuki&kasami distributed mutual exclusion algorithm. In *FMOODS'02*, pages 181–195. Kluwer Academic Publishers, 2002.

[10] K. Ogata and K. Futatsugi. Rewriting-based verification of authentication protocols. In *Proceedings of WRLA'02*, volume 71 of ENTCS. Elsevier Science Publishers, 2002.

[11] K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ method. In *the 6th IFIP WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, LNCS 2884. Springer, pp.170–184, 2003.

[12] T. Seino, K. Ogata, and K. Futatsugi. Specification and verification of a single-track railroad signaling in CafeOBJ. *IEICE TRANSACTIONS on Fundamental of Electronics, Communications and Computer Science*, E84-A(6):1471–1478, 2001.

[13] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, em Formal Methods Europe '93: Industrial-Strength Formal Methods, pages 357–406, Odense, Denmark, April 1993.

[14] T. N. Win, M. D. Ernst, S. J. Garland, D. Kirli, and N. A. Lynch. Using simulated execution in verifying distributed algorithms. In *VMCAI'03, Fourth International Conference on Verification, Model Checking and Abstract Interpretation*, pages 283–297, 2003.