A Toolkit for Generating and Displaying Proof Scores in the OTS/CafeOBJ Method ⋆

Takahiro Seino [1]

Japan Advanced Institute of Science and Technology (JAIST)

Kazuhiro Ogata [2]

NEC Software Hokuriku, Ltd. / JAIST

Kokichi Futatsugi [3]

Japan Advanced Institute of Science and Technology (JAIST)

Abstract

The OTS/CafeOBJ method can be used to model, specify and verify distributed systems. Specifications are written in equations, which are regarded as rewrite rules and used to verify specifications. The usefulness of the method is demonstrated by applying the method to nontrivial problems such as electronic commerce protocols and railroad signaling systems. In this paper we describe a toolkit called Buffet, which assists verification in the method. Given predicates used to split cases and lemmas, Buffet automatically generates proofs (called proof scores) and checks the proof scores using the CafeOBJ system. Buffet also has facilities to display proof scores generated and verification results on a web browser.

1 Introduction

Abstract machines as well as abstract data types can be specified in CafeOBJ[4], an algebraic specification language. Algebraic specifications of abstract machines are called behavioral specifications. Behavioral specifications are written in equations, which are regarded as rewrite rules and used to

⋆ 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.

[1] Email: t-seino@jaist.ac.jp

[2] Email: ogatak@acm.org

[3] Email: kokichi@jaist.ac.jp

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

verify behavioral specifications. Rewriting is an efficient way of implementing equational reasoning, which is the most fundamental way of reasoning and can moderate the difficulties of proofs that might otherwise become too hard to understand.

We use observational transition systems (OTSs; which are transition systems that can be straightforwardly written in equations) as abstract machines and have been developing a method of verifying behavioral specifications. The method is called the OTS/CafeOBJ method[15]. In the OTS/CafeOBJ method, a system is modeled as an OTS, the OTS is written in CafeOBJ and it is verified that the OTS has properties by writing proofs (called proof scores) in CafeOBJ and checking the proof scores by means of rewriting. We have been demonstrating its usefulness by doing case studies, among which are [13,14,17]. In the case studies, however, basically proof scores were entirely written by hand using usual text editors such as Emacs, which is subject to human errors such that some cases to consider may be overlooked.

We have then designed and implemented a toolkit called Buffet, which assists verification in the OTS/CafeOBJ method. Given predicates used to split cases and lemmas, Buffet automatically generates proof scores and checks the proof scores using the CafeOBJ system. Although the success of a proof depends on given predicates for case analysis and lemmas, it is guaranteed that generated proof scores cover all cases, excluding human errors. Buffet also has facilities to display proof scores generated and verification results on a web browser. Since Buffet only displays by default parts of a proof score hierarchically for which further case analysis should be done and/or lemmas should be used, the facilities can help users find how to split cases and what lemmas to use. The facilities can also help users read and understand proof scores. In this paper we describe Buffet and report on a case study that Buffet has been applied to a simple mutual exclusion protocol.

2 Preliminaries

We assume that there exists a universal state space denoted by Υ and data types used, including the equivalence relation denoted by $=$ for each data type, have been defined. An OTS[15] $\mathcal{S}$ consists of $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that 1) $\mathcal{O}$: 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 $\upsilon_1, \upsilon_2 \in \Upsilon$, the equivalence $(\upsilon_1 =_{\mathcal{S}} \upsilon_2)$ between them wrt $\mathcal{S}$ is defined as $\forall o \in \mathcal{O}.o(\upsilon_1) = o(\upsilon_2)$, 2) $\mathcal{I}$: the set of initial states such that $\mathcal{I} \subseteq \Upsilon$, and 3) $\mathcal{T}$: a set of conditional transitions; each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon \to \Upsilon$ such that $\tau(\upsilon_1) =_{\mathcal{S}} \tau(\upsilon_2)$ for each $[\upsilon] \in \Upsilon/=_{\mathcal{S}}$ and each $\upsilon_1, \upsilon_2 \in [\upsilon]$; $\tau(\upsilon)$ is called the successor state of $\upsilon \in \Upsilon$ wrt τ; the condition c_τ of τ is called the effective condition. An execution of $\mathcal{S}$ is an infinite sequence $\upsilon_0, \upsilon_1, \ldots$ of states satisfying *Initiation* ($\upsilon_0 \in \mathcal{I}$) and *Consecution* ($\forall i \in \{0, 1, \ldots\}. \exists \tau \in \mathcal{T}.(\upsilon_{i+1} =_{\mathcal{S}} \tau(\upsilon_i))$). A state υ is called reachable wrt $\mathcal{S}$ iff there exists an execution of $\mathcal{S}$ in which υ appears.

Properties discussed in this paper are invariants only. A predicate p is called invariant wrt $\mathcal{S}$ iff $p(v)$ holds for every reachable state v wrt $\mathcal{S}$. Observers and transitions may be parameterized, which are generally expressed as $o_{i_1,\ldots,i_m} : \Upsilon \to D_i$ and $\tau_{j_1,\ldots,j_n} : \Upsilon \to \Upsilon$, provided that $m, n \geq 0$ and there exists a data type D_k such that $k \in D_k$ for $k = i_1, \ldots, i_m, j_1, \ldots, j_n$.

CafeOBJ[4] (see `www.ldl.jaist.ac.jp/cafeobj/`) is an algebraic specification language/system mainly based on order-sorted algebras[6] and hidden-sorted algebras[5,9]. Abstract machines as well as abstract data types can be specified in CafeOBJ, which has two kinds of sorts: visible and hidden sorts denoting abstract data types and the state spaces of abstract machines, and two kinds of operators wrt hidden sorts: action and observation operators that denote state transitions of abstract machines and let us know the state of abstract machines. Both an action operator and an observation operator take a state of an abstract machine and zero or more data, an action operator returns the successor state and an observation operator returns a value that characterizes the state of an abstract machine. The syntax of operator declarations is

[`b`]`op` *OpName* `:` *Sort** `->` *Sort*

`bop` is used for action and observation operators, while `op` for others. Operators are defined with equations. The syntax of equation declarations is

[`c`]`eq` *Term* `=` *Term* [`if` *Term*] `.`

`ceq` is used for conditional equations, while `eq` for non-conditional ones. The CafeOBJ system uses equations as rewrite rules and rewrites terms. CafeOBJ is also based on rewriting logic. The syntax of rewriting rules is

`trans` *Term* `=>` *Term* `.`

In Buffet, rewriting rules are used to instruct Buffet to generate proof scores.

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 visible sort `Bool` denoting truth values, 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 conditional choice operator `if_then_else_fi` 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 rewrite rules, they are complete wrt propositional logic. Therefore, any term denoting a propositional formula that is always true (or false) surely reduces to `true` (or `false`). Generally, a term of `Bool` reduces to an exclusive

disjunction of conjunctions.

$\mathcal{S}$ is written in CafeOBJ. Υ is denoted by a hidden sort, say H, $o_{d_{i_1},\ldots,d_{i_m}}$ by a CafeOBJ observation operator, say o, and $\tau_{d_{j_1},\ldots,d_{j_n}}$ by a CafeOBJ action operator, say a; o and a are declared as

`bop` $o : H\ V_{i_1}\ \ldots\ V_{i_m}$ `->` V_i `bop` $a : H\ V_{j_1}\ \ldots\ V_{j_n}$ `->` H

V_k is a visible sort corresponding to D_k for $k = i_1, \ldots, i_m, i, j_1, \ldots, j_n$. Any state in $\mathcal{I}$, i.e. any initial state, is denoted by a constant, say *init* declared as

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

We suppose that the initial value of each $o_{i_1,\ldots,i_m}$ is $f(i_1, \ldots, i_m)$. The initial value of each $o_{i_1,\ldots,i_m}$ is specified with the equation

`eq` $o(\mathit{init}, X_{i_1}, \ldots, X_{i_m}) = f(X_{i_1}, \ldots, X_{i_m})$.

X_k is a CafeOBJ variable of sort V_k for $k = i_1, \ldots, i_m$ and $f(X_{i_1}, \ldots, X_{i_m})$ is a CafeOBJ term denoting $f(i_1, \ldots, i_m)$. Each $\tau_{j_1,\ldots,j_n}$ may change the value of each $o_{i_1,\ldots,i_m}$ if it is applied in a state υ such that $c_{\tau_{j_1,\ldots,j_n}}$ holds, which can be written as

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

S is a CafeOBJ variable of H and each X_k is a CafeOBJ variable of V_k. $a(S, X_{j_1}, \ldots, X_{j_n})$ denotes the successor state of S wrt $\tau_{j_1,\ldots,j_n}$. $e\text{-}a(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})$ denotes the value of $o_{i_1,\ldots,i_m}$ in the successor state. $c\text{-}a(S, X_{j_1}, \ldots, X_{j_n})$ denotes $c_{\tau_{j_1,\ldots,j_n}}$. $\tau_{j_1,\ldots,j_n}$ changes nothing if it is applied in a state υ such that $c_{\tau_{j_1,\ldots,j_n}}$ does not hold, which can be written as

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

If the value of $o_{i_1,\ldots,i_m}$ is not affected by applying $\tau_{j_1,\ldots,j_n}$ in any state (regardless of the truth value of $c_{\tau_{j_1,\ldots,j_n}}$), the following equation may be declared:

`eq` $o(a(S, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m}) = o(S, X_{i_1}, \ldots, X_{i_m})$.

3 Proof Scores

We describe proof scores showing that a predicate p_1 is invariant wrt $\mathcal{S}$, which are written in CafeOBJ. We often need other predicates, say $p_2, \ldots, p_n$, for the verification, although such predicates should be found during the verification. Let $x_{i1}, \ldots, x_{im_i}$, whose types are $D_{i1}, \ldots, D_{im_i}$, be all free variables in p_i except υ, whose type is Υ, for $i = 1, \ldots, n$. p_i may be written as $p_i(\upsilon, x_{i1}, \ldots, x_{im_i})$.

Although some invariant properties may be proved by rewriting and case analysis only with other proved invariant properties, we often need induction, especially simultaneous induction[15] on the number of transitions applied.

We first declare the operators denoting $p_1, \ldots, p_n$ and the equations defining the operators. The operators and equations are declared in a module, say

`INV` (which imports the module where $\mathcal{S}$ is written), as

`op` inv_i : H $V_{i1} \ldots V_{im_i}$ `->` `Bool`
`eq` $inv_i(S, X_{i1}, \ldots, X_{im_i}) = p_i(S, X_{i1}, \ldots, X_{im_i})$.

for $i = 1, \ldots, n$. V_k is a visible sort denoting D_k and X_k is a CafeOBJ variable of V_k for $k = i1, \ldots, im_i$. $p_i(S, X_{i1}, \ldots, X_{im_i})$ is a CafeOBJ term denoting p_i. In module `INV`, we also declare a constant x_k denoting an arbitrary value of V_k for $k = 1, \ldots, n$.

We then declare the operators denoting basic formulas to show in the inductive cases and the equations defining the operators. The operators and equations are declared in a module, say `ISTEP` (which imports `INV`), as follows:

`op` $istep_i$: $V_{i1} \ldots V_{im_i}$ `->` `Bool`
`eq` $istep_i(X_{i1}, \ldots, X_{im_i}) = inv_i(s, X_{i1}, \ldots, X_{im_i})$ `implies` $inv_i(s', X_{i1}, \ldots, X_{im_i})$.

for $i = 1, \ldots, n$. s and s', which are declared in module `ISTEP`, are constants of H; s denotes an arbitrary state and s' a successor state of the state.

For the base case, we write

```
open INV
 red invi(init, xi1, ..., ximi) .
close
```

for $i = 1, \ldots, n$. CafeOBJ command `open` makes a temporary module that imports a module given as an argument and CafeOBJ command `close` destroys the temporary module. Parts enclosed with `open` and `close` are basic units of proof scores, which are called proof passages in the OTS/CafeOBJ method.

For the induction case showing that each $\tau_{j_1,\ldots,j_{m_j}}$ (denoted by action operator a) preserves each p_i, we often need case analysis. We suppose that the state space is split into l sub-spaces for the induction case, although such case analysis should be done during the verification. Each of the l subcases is supposed to be characterized by a predicate $case_{i_k}$ for $k = 1, \ldots, l$; the predicates should satisfy $(case_{i_1} \vee \ldots \vee case_{i_l}) \Leftrightarrow$ true. For the induction case, we then write

```
open ISTEP
 -- arbitrary objects
 op y1m1 : -> V1m1 . ... op yjmj : -> Vjmj .
 -- assumptions
 Declaration of equations denoting case_ik.
 -- successor state
 eq s' = a(s, yj1, ..., yjmj) .
 -- check
 red SIHi implies istepi(xi1, ..., ximi) .
close
```

for $i = 1, \ldots, n$ and $k = 1, \ldots, l$. A comment starts with `--` and terminates at the end of the line. SIH_i is used to strengthen the induction

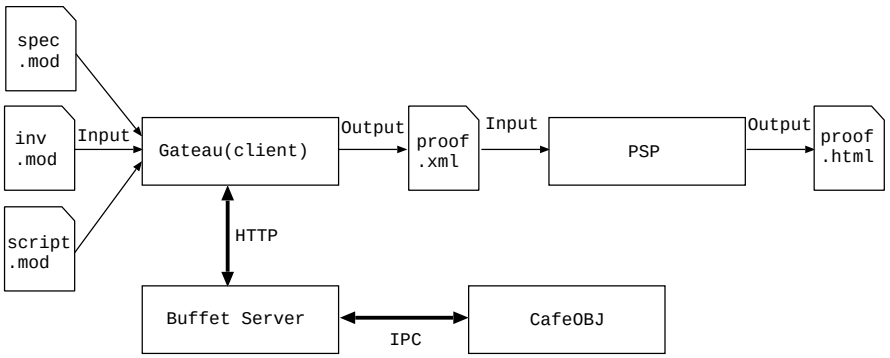

Fig. 1. An overview of the Buffet toolkit.

hypothesis $inv_i(s, x_{i1}, \ldots, x_{im_i})$ and is the form $inv_{\iota_1}(s, t_{\iota_1 1}, \ldots)$ `and` ... `and` $inv_{\iota_\kappa}(s, t_{\iota_\kappa 1}, \ldots)$, where $1 \leq \iota_1, \ldots, \iota_\kappa \leq n$ and each t_k is a term of sort V_k.

4 Buffet: A Toolkit for the OTS/CafeOBJ Method

Buffet is a toolkit for generating and displaying proof scores. An overview of Buffet is shown in Fig. 1. Buffet consists of the Buffet server, Gateau (a Buffet client), PSP (Proof Score Presenter) and the CafeOBJ system. Gateau takes three kinds of files: *spec.mod* in which an OTS $\mathcal{S}$ is specified in CafeOBJ, *inv.mod* in which modules `INV` and `ISTEP` are declared, and *script.mod* in which a script to instruct Gateau to generate a proof score is written. Gateau communicates with the CafeOBJ system via the Buffet server using the HTTP protocol and an inter-process communication (IPC) method. Given *spec.mod*, *inv.mod* and *script.mod*, then Gateau feeds them into the CafeOBJ system, generates a proof score based on some information extracted from the three files and passes the proof score into the CafeOBJ system. Gateau then receives the results of rewriting the proof score from the CafeOBJ system, generates a file *proof.xml* in XML from the proof score and the results, and passes the file to PSP. PSP then generates a file *proof.html* in HTML from *proof.xml* to display the proof score and the results on a web browser. In the rest of the section, we describe the Buffet server, Gateau and PSP.

4.1 The Buffet Server

The Buffet server provides the five services: 1) to create a new session, 2) to have the CafeOBJ system load files, 3) to obtain a module information from the CafeOBJ system, reconstruct the module as an XML document and pass it to a client, 4) to have the CafeOBJ system reduce a term under a module and pass the result reconstructed as an XML document to a client, and 5) to finish the current session.

We describe the five services in turn. When requested by a client, the Buffet server creates a new session for the client, starting the CafeOBJ system as its child process and establishing an IPC connection between the server

process and the child process. After that, the client can communicate with the CafeOBJ system via the Buffet server. CafeOBJ command `in` is used to load files into the CafeOBJ system. The Buffet server has the CafeOBJ system load a file by sending command `in` and the file name to the CafeOBJ system. CafeOBJ command `show` is used to parse modules and use the results of parsing them, provided that switch `tree print` is set to `on`. The Buffet server has the CafeOBJ system parse a module by sending command `show` and the module name to the CafeOBJ system, reconstructs the module as an XML document based on the result of parsing the module and passes it to a client. The Buffet server has the CafeOBJ system reduce a term under a module by sending command `red`, the term and the module name, reconstructs the result term as an XML document and passes it to a client. At the end of a session, the Buffet server finishes the session by stopping the CafeOBJ system.

The current implementation of the Buffet server is written in Perl and consists of about 1,200 lines.

4.2 Gateau

Gateau has the five commands `new`, `input`, `parse`, `verify` and `quit`. Commands `new` and `quit` correspond to the first and fifth services provided by the Buffet server. Command `input` takes a file name as its argument and uploads the file to the Buffet server, which does the second service. Command `parse` takes the name of a module in which an OTS $\mathcal{S}$ is supposed to be specified and passes it to the Buffet server, which does the third service; Gateau extracts the action operators denoting the transitions of $\mathcal{S}$ and their (effective) conditions from the module (which is an XML document) returned by the Buffet server. The action operators and their conditions are used to generate proof scores showing that predicates are invariant wrt $\mathcal{S}$. Command `verify` takes the name of a module in which a proof script is supposed to be written for the proof that a predicate is invariant wrt $\mathcal{S}$ and passes it to the Buffet server, which does the third service; Gateau generates a proof score based on the module (which is an XML document) returned by the Buffet server and the information obtained by command `parse`. For each proof passage of the proof score, Gateau makes a module corresponding to the proof passage (excluding the statement containing CafeOBJ command `red`), uploads the module to the Buffet server so as to have the CafeOBJ system load the module, asks the Buffet server to has the CafeOBJ system reduce the term (appearing in the proof passage and denoting the formula to be proved) under the module, and receives the result (which is an XML document) from the CafeOBJ system via the Buffet server. Based on the results and the proof score, Gateau makes an XML document of them, saves it as a file and passes the file name to PSP.

The current implementation of Gateau is written in Perl and consists of about 1,400 lines. In the rest of this subsection, we describe proof scripts and how to generate proof scores based on proof scripts.

4.2.1 Proof Scripts

For each predicate p_i (denoted by operator inv_i) to be verified, we write a proof script from which a proof score of p_i is generated. In a proof script of p_i, for each action operator a denoting transition $\tau_{j_1,\ldots,j_{m_j}}$, we give predicates such as $c_{i_1},\ldots,c_{i_{n_i}}$ that are used to split cases and formulas such as $inv_{\iota_1}(s,t_{\iota_1 1},\ldots),\ldots\ inv_{\iota_\kappa}(s,t_{\iota_\kappa 1},\ldots)$ that are used to strengthen the induction hypothesis $inv_i(s,x_{i1},\ldots,x_{im_i})$ for the induction case that $\tau_{j_1,\ldots,j_{m_j}}$ preserves p_i. Such predicates and formulas are given in the form of rewriting rules. The rewriting rules for predicates $c_{i_1},\ldots,c_{i_{n_i}}$ and formulas $inv_{\iota_1}(s,t_{\iota_1 1},\ldots),\ldots\ inv_{\iota_\kappa}(s,t_{\iota_\kappa 1},\ldots)$ look like

`trans predicates`$(a(S,Y_{j_1},\ldots,Y_{j_{m_j}}))$ `=>` c_{i_1} .
...
`trans predicates`$(a(S,Y_{j_1},\ldots,Y_{j_{m_j}}))$ `=>` $c_{i_{n_i}}$.
`trans lemmas`$(a(S,Y_{j_1},\ldots,Y_{j_{m_j}}))$ `=>` $inv_{\iota_1}(s,t_{\iota_1 1},\ldots)$.
...
`trans lemmas`$(a(S,Y_{j_1},\ldots,Y_{j_{m_j}}))$ `=>` $inv_{\iota_\kappa}(s,t_{\iota_\kappa 1},\ldots)$.

Operators `predicates` and `lemmas` are used as keywords to write such predicates and formulas. The reason why such predicates and formulas are given in the form of rewriting rules is that we can use the CafeOBJ systems to parse rewriting rules and do not have to implement another parser for such predicates and formulas. Proof scores are generated based on such predicates and formulas, which is next described.

4.2.2 How to Generate Proof Scores

Given predicate p_i (denoted by operator inv_i) to be verified, predicates c_{i_1}, $\ldots,c_{i_{n_i}}$ used to split cases and formulas $inv_{\iota_1}(s,t_{\iota_1 1},\ldots),\ldots\ inv_{\iota_\kappa}(s,t_{\iota_\kappa 1},\ldots)$ used to strengthen the induction hypothesis $inv_i(s,x_{i1},\ldots,x_{im_i})$, then a proof score of p_i is generated and checked as follows:

(i) Base case: Gateau has the CafeOBJ system reduce term $inv_i(init,x_{i1},\ldots,x_{im_i})$ under module `INV` and generates an XML document of the proof passage and the result.

(ii) Induction cases: For each action operator a denoting transition $\tau_{j_1,\ldots,j_{m_j}}$, for each proof passage to be checked a temporary module `PROOF_TMP` is generated and a term denoting a formula to be proved is reduced under `PROOF_TMP` in the following way (let c_{i_0} be $c_{\tau_{j_1,\ldots,j_{m_j}}}$):

$k := 1$; $stack := empty$; $push(stack,\{c_{i_0}\})$; $push(stack,\{\neg c_{i_0}\})$;
while $stack \neq empty$ **do**
  $Cs := pop(stack)$; (* let Cs be $\{c'_1,\ldots,c'_{n'}\}$. *)
  Make the module
   `mod PROOF_TMP {`
    `pr(ISTEP)`
    `op` $y_{1_{m_1}}$ `: ->` $V_{1_{m_1}}$ . ... `op` $y_{j_{m_j}}$ `: ->` $V_{j_{m_j}}$.
    Declaration of equations denoting $c'_1,\ldots,c'_{n'}$.

```
    eq s' = a(s, y_{j_1}, ..., y_{j_{m_j}}) .
  } ;
  Let T be istep_i(x_{i1}, ..., x_{im_i});
  Have the CafeOBJ system reduce T under PROOF_TMP;
  if the result is true then
    (* The proof succeeds in the case Cs. *)
    Generate an XML document of the proof passage and the result;
  else if the result is false then
    Let T be SIH_i implies istep_i(x_{i1}, ..., x_{im_i});
    Have the CafeOBJ system reduce T under PROOF_TMP;
    Generate an XML document of the proof passage and the result;
    (* If the result is true, the proof succeeds in the case Cs. *)
    (* If not, other lemmas may be needed. *)
  else if k ≤ i_{n_i} then
    push(stack, Cs ∪ {c_k}); push(stack, Cs ∪ {¬c_k}); k := k + 1;
  else
    Let T be SIH_i implies istep_i(x_{i1}, ..., x_{im_i});
    Have the CafeOBJ system reduce T under PROOF_TMP;
    Generate an XML document of the proof passage and the result;
    (* If the result is true, the proof succeeds in the case Cs. *)
    (* If not, further case analysis and/or other lemmas may be needed. *)
  fi fi fi;
od
```

k is an integer variable and *stack* is a stack of predicate sets. *push* and *pop* are usual operators of stacks. `mod` is the keyword for declaring modules and `pr` is the keyword for importing modules.

Each predicate $c'_{k'}$ is the form $l_1 \wedge \ldots l_{n'_{k'}}$ or $\neg(l_1 \wedge \ldots l_{n'_{k'}})$, where each l_κ is a literal, namely the form α_κ or $\neg\alpha_\kappa$, and α_κ is an atomic formula. In the case that $c'_{k'}$ is the form $\neg(l_1 \wedge \ldots l_{n'_{k'}})$, we declare equation $c'_{k'} = \texttt{false}$. In the case that $c'_{k'}$ is the form $l_1 \wedge \ldots l_{n'_{k'}}$, we declare an equation for each l_κ. In the case that l_κ is the form $\neg\alpha_\kappa$, we declare equation $l_\kappa = \texttt{false}$. In the case that l_κ is the form α_κ, if α_κ is the form $left = right$, we declare equation $left = right$, and otherwise we declare equation $l_\kappa = \texttt{true}$.

4.3 PSP

Given an XML document of a proof score and the results of reducing the proof passages in the proof score, PSP generates an HTML document. When an HTML document generated by PSP is first displayed on a web browser, proof passages for which results are not `true` and their results are shown, and other proof passages (for which the proof has succeeded) are hidden. Proof passages are hierarchically shown according to the predicates used to split cases and each proof passage is clickable, allowing the proof passage to appear and disappear. The current implementation of PSP is written in XSLT (XSL Transformations; see `www.w3.org/TR/xslt`) and consists of about 600 lines.

5 A Case Study: A Mutual Exclusion Protocol

We describe a case study that Buffet has been applied to the verification that a simple mutual exclusion protocol has the mutual exclusion property. The protocol repeatedly executed by multiple processes can be written as

l1: Remainder Section
l2: **repeat until** $\neg$*fetch&store*(*lock*, true)
  Critical Section
cs: *lock* := false

lock is a boolean variable and is initially set to false. *fetch&store*(x, v) atomically exchanges the value of variable x with value v and returns the original value of x. Each process is initially at location l1.

5.1 Modeling and Specification of the Mutual Exclusion Protocol

Let B, P and L be types of boolean values, process IDs and locations (l1, l2 and cs). The mutual exclusion protocol is modeled as the OTS $\mathcal{S}_{MX}$ such that 1) $\mathcal{O}_{MX}$ consists of $lock : \Upsilon \to B$ and $loc_i : \Upsilon \to L$ for $i \in P$, 2) $\mathcal{I}_{MX}$ is $\{\upsilon \in \Upsilon \mid \neg lock(\upsilon) \wedge \forall i \in P.(loc_i(\upsilon) = \mathrm{l1})\}$, and 3) $\mathcal{T}_{MX}$ consists of $try_i : \Upsilon \to \Upsilon$, $enter_i : \Upsilon \to \Upsilon$ and $leave_i : \Upsilon \to \Upsilon$, for $i \in P$, whose effective conditions are $c_{try_i}(\upsilon) \equiv (loc_i(\upsilon) = \mathrm{l1})$, $c_{enter_i}(\upsilon) \equiv (loc_i = \mathrm{l2} \wedge \neg lock(\upsilon))$ and $c_{leave_i}(\upsilon) \equiv (loc_i = \mathrm{cs})$ and whose definitions are:

(i) Let υ' be $try_i(\upsilon)$. If $c_{try_i}(\upsilon)$ holds, then $lock(\upsilon') = lock(\upsilon)$ and $loc_j(\upsilon') =$ (**if** $i = j$ **then** l2 **else** $loc_j(\upsilon)$). Otherwise, nothing changes.

(ii) Let υ' be $enter_i(\upsilon)$. If $c_{enter_i}(\upsilon)$ holds, then $lock(\upsilon') = \mathrm{true}$ and $loc_j(\upsilon') =$ (**if** $i = j$ **then** cs **else** $loc_j(\upsilon)$). Otherwise, nothing changes.

(iii) Let υ' be $leave_i(\upsilon)$. If $c_{leave_i}(\upsilon)$ holds, then $lock(\upsilon') = \mathrm{false}$ and $loc_j(\upsilon') =$ (**if** $i = j$ **then** l1 **else** $loc_j(\upsilon)$). Otherwise, nothing changes.

$\mathcal{S}_{MX}$ is written in CafeOBJ. The signature of the CafeOBJ specification of $\mathcal{S}_{MX}$ is

```
-- any initial state
op init : -> Sys
-- observation operators
bop lock : Sys -> Bool        bop loc  : Sys Pid -> Loc
-- action operators
bop try   : Sys Pid -> Sys    bop enter : Sys Pid -> Sys
bop leave : Sys Pid -> Sys
```

`Sys` is the hidden sort denoting Υ, `Pid` is the visible sort denoting P and `Loc` is the visible sort denoting L. Constant `init` denotes any initial state. Observation operators `lock` and `loc` denote observers *lock* and loc_i, and action operators `try`, `enter` and `leave` denote transitions try_i, $enter_i$ and $leave_i$. The three action operators are defined in equations as

```
-- try
```

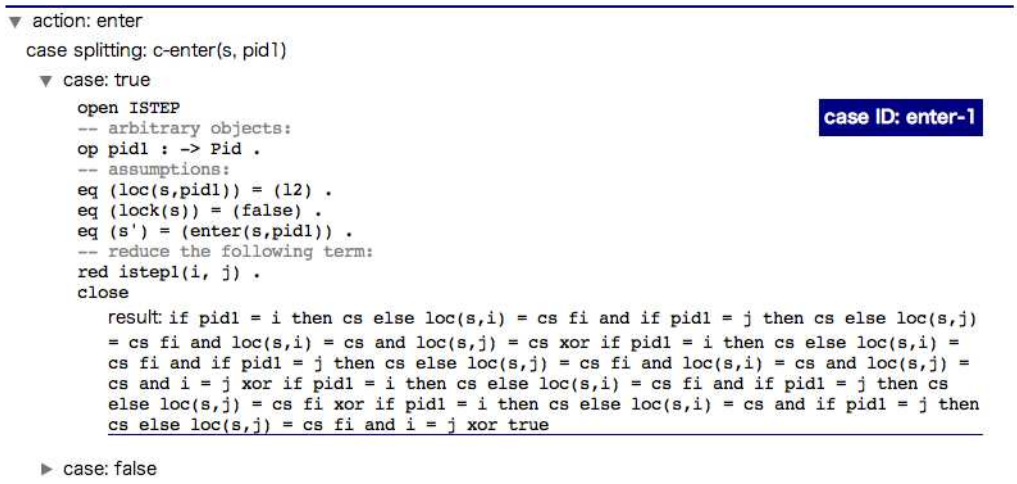

Fig. 2. Excerpts from the proof score and the results displayed by Buffet (1).

```
eq  lock(try(S,I))     = lock(S) .
ceq loc(try(S,I),J)    = (if I = J then l2 else loc(S,J) fi)
                              if c-try(S,I) .
ceq try(S,I)           = S     if not (c-try(S,I)) .
-- enter
ceq lock(enter(S,I))   = true  if c-enter(S,I) .
ceq loc(enter(S,I),J)  = (if I = J then cs else loc(S,J) fi)
                              if c-enter(S,I) .
ceq enter(S,I)         = S     if not(c-enter(S,I)) .
-- leave
ceq lock(leave(S,I))   = false if c-leave(S,I) .
ceq loc(leave(S,I),J)  = (if I = J then l1 else loc(S,J) fi)
                              if c-leave(S,I) .
ceq leave(S,I)         = S     if not (c-leave(S,I)) .
```

Operators `c-try`, `c-enter` and `c-leave` denote c_{try_i}, c_{enter_i} and c_{leave_i}, which are defined as

```
eq c-try(S,I) = (loc(S,I) = l1) .
eq c-enter(S,I) = (loc(S,I) = l2 and not lock(S)) .
eq c-leave(S,I) = (loc(S,I) = cs) .
```

5.2 Verification of the Mutual Exclusion Protocol

We describe the verification that $\mathcal{S}_{MX}$ has the mutual exclusion property. For the verification, all we have to do is to prove predicate $(loc_i(v) = \mathrm{cs} \wedge loc_j(v) = \mathrm{cs}) \Rightarrow (i = j)$ invariant wrt $\mathcal{S}_{MX}$. The predicate is denoted by operator `inv1` defined as

```
eq inv1(S,I,J) = (loc(S,I) = cs and loc(S,J) = cs implies I = J) .
```

The operator is declared and defined in module `INV`. In the module, constants `i` and `j` denoting arbitrary values of sort `Pid` are also declared. The operator

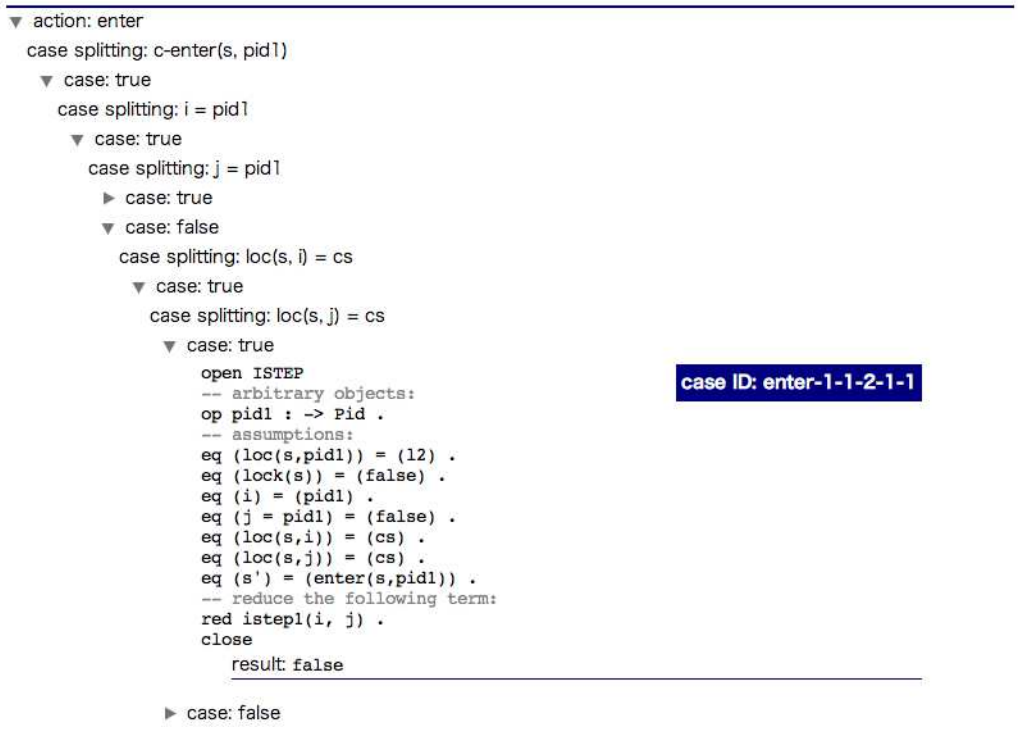

Fig. 3. Excerpts from the proof score and the results displayed by Buffet (2).

denoting the basic formula to be shown in each induction case is denoted by operator `istep1` defined as

```
eq istep1(I,J) = inv1(s,I,J) implies inv1(s',I,J) .
```

The operator is declared and defined in module `ISTEP`. In the module, constants `s` and `s'` are also declared.

First of all we do not use any predicates to split cases and any formulas to strengthen the induction hypothesis and have Buffet generate and check a proof score of `inv1(s,i,j)` and display the proof score and the results. Buffet reports that seven cases have been checked and the proof has succeeded in three out of the seven cases. We show in Fig. 2 part of the proof score and the results displayed by Buffet. Small triangles are clickable buttons. An upside down triangle means that its contents are shown, and a triangle rotated clockwise by 90 degrees means that its contents are hidden. The first button from top in Fig. 2 corresponds to the induction case of $enter_i$ denoted by `enter`. There are two proof passages in the induction case. One corresponding to the second button is shown and the other corresponding to the last button is hidden. The proof succeeds in the second proof passage but for the first proof passage we need case analysis.

Next we use predicates to split cases and the predicates are given as

```
-- for try
trans predicates(try(S,P)) => (i = pid1) .
trans predicates(try(S,P)) => (j = pid1) .
```

```
-- for enter
trans predicates(enter(S,P)) => (i = pid1) .
trans predicates(enter(S,P)) => (j = pid1) .
trans predicates(enter(S,P)) => (loc(s,i) = cs) .
trans predicates(enter(S,P)) => (loc(s,j) = cs) .
-- for leave
trans predicates(leave(S,P)) => (i = pid1) .
trans predicates(leave(S,P)) => (j = pid1) .
```

But we do not use any formulas to strengthen the induction hypothesis. In this case Buffet reports that 18 cases have been checked and the proof has succeeded in 15 out of the 18 cases. We show in Fig. 3 part of the proof score and the results displayed by Buffet.

Looking at the proof passage in Fig. 3, we notice that process `j` is at location `cs` and `lock(s)` is `false` in state `s`, which seems contradiction. Therefore we conjecture that predicate $(loc_i(v) = \mathrm{cs}) \Rightarrow lock(v)$ is also invariant. The predicate is denoted by operator `inv2` defined (in module `INV`) as

```
eq inv2(S,I) = (loc(S,I) = cs implies lock(S)) .
```

We also declare and define operator `istep2` denoting the basic formula to be shown in each induction case as `istep1` in module `ISTEP`.

In addition to the predicates to split cases, we also use formulas to strengthen the induction hypothesis `inv1(s,i,j)` and the formulas are given as

```
trans lemmas(enter(S,P)) => inv2(s,i) .
trans lemmas(enter(S,P)) => inv2(s,j) .
```

In this case Buffet reports that the proof has succeeded in all 18 cases.

For the verification of `inv2(s,i)`, we use the predicates to split cases and the formulas to strengthen the induction hypothesis:

```
-- for try
trans predicates(try(S,P)) => (pid1 = i) .
trans predicates(try(S,P)) => (loc(s,i) = cs) .
trans predicates(try(S,P)) => lock(s) .
trans predicates(try(S,P)) => (loc(s,i) = l1) .
-- for leave
trans predicates(leave(S,P)) => (pid1 = i) .
trans predicates(leave(S,P)) => (loc(s,i) = cs) .
trans predicates(leave(S,P)) => lock(s) .
trans lemmas(leave(S,P)) => inv1(s,i,pid1) .
```

Buffet reports that 15 cases have been checked and the proof has succeeded in all the cases.

Note that the predicate denoted by `inv2` is needed to strengthen the induction hypothesis for the invariant proof of the predicate denoted by `inv1` and vice versa, which means that if each of the proof scores is written individually, then simultaneous induction[15] is needed.

6 Related Work

BOBJ[8] is an algebraic specification language based on order-sorted and hidden-sorted algebras, which is a sibling language of CafeOBJ. BOBJ implements conditional circular coinductive rewriting with case analysis (c4rw). Given equations (which are used to split cases) and lemmas, c4rw automatically generates proof scores and checks the proof scores. BOBJ allows us to specify how to split cases in more detail than Buffet, but users who give equations used to split cases are responsible for whether the whole cases are covered by the equations. As shown in [8], some problems can be verified well with c4rw, but it seems that further research should be done to make it clear that BOBJ can be appropriately applied to what types of problems. BOBJ is part of the Tatami system[7]. The Tatami system provides facilities for displaying proofs so as to make them preferably attractive to software engineers based on algebraic semiotics (which combines algebraic specification with social semiotics). The basic idea behind the facilities may be used to improve our way of displaying proof scores.

Several proof assistants have been proposed. Among them are Coq[1] and Isabell/HOL[12]. They provide 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 zero or more proof sub-goals, which are hopefully simpler. But users should select appropriate tactics in order to succeed in their proofs. This means that users are basically required to have knowledge and experience to complete their proofs on their own without any proof assistants, although proof assistants prevent users from making mistakes.

Among the existing tools supporting verification of (distributed) systems with algebraic specification languages are Larch Prover (LP)[10] and Maude Inductive Theorem Prover (Maude ITP)[3]. 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. Maude ITP assists verification of abstract data types written in Maude[2], an algebraic specification and programming language based on membership equational logic and rewriting logic, but does not assist verification of abstract machines.

7 Conclusion

We have described Buffet for generating and displaying proof scores in the OTS/CafeOBJ method and reported on the case study on the verification of a simple mutual exclusion protocol. In addition to the mutual exclusion protocol, Buffet has been successfully applied to the verification that the NSLPK authentication protocol[11] and the Otway-Rees authentication protocol[16] have the secrecy property.

References

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

[2] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. Quesada, *Maude: Spesification and programming language in rewriting logic*, TCS **285** (2002), pp. 187–243.

[3] Clavel, M., F. Durán, S. Eker and J. Meseguer, *Building equational proving tools by reflection in rewriting logic*, in: *CAFE: An Industrial-Strength Algebraic Formal Method*, Elsevier, 2000 pp. 1–31.

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

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

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

[7] Goguen, J. A. and K. Lin, *Web-based support for cooperative software engineering*, Annals of Software Engineering **12** (2001), pp. 167–191.

[8] Goguen, J. A. and K. Lin, *Behavioral verification of distributed concurrent systems with BOBJ*, in: *3rd QSIC* (2003), pp. 216–235.

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

[10] Guttag, J. V., 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] Lowe, G., *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*, in: *2nd TACAS*, LNCS 1055 (1996), pp. 147–166.

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

[13] Ogata, K. and K. Futatsugi, *Formal analysis of Suzuki&Kasami distributed mutual exclusion algorithm*, in: *5th FMOODS* (2002), pp. 181–195.

[14] Ogata, K. and K. Futatsugi, *Formal analysis of the iKP electronic payment protocols*, in: *1st ISSS*, LNCS **2609** (2003), pp. 441–460.

[15] Ogata, K. and K. Futatsugi, *Proof scores in the OTS/CafeOBJ method*, in: *6th FMOODS*, LNCS **2884** (2003), pp. 170–184.

[16] Otway, D. and O. Rees, *Efficient and timely mutual authentication*, ACM Operating Systems Review **21** (1987), pp. 8–10.

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