Verifying Design with Proof Scores

Kokichi FUTATSUGI[1], Joseph A. GOGUEN[2], and Kazuhiro OGATA[3]

[1] Japan Advanced Institute of Science and Technology (JAIST)
futatsugi@jaist.ac.jp
[2] University of California at San Diego
goguen@cs.ucsd.edu
[3] NEC Software Hokuriku, Ltd. and JAIST
ogata@jaist.ac.jp

Abstract: Verifying design instead of code can be an effective and practical approach to obtaining verified software. This paper argues that proof scores are an attractive method for verifying design, in that they achieve a balance in which the respective capabilities of humans and machines are utilized optimally.

1 Verifying Code or Design

Although creation of a verifying compiler is a difficult challenge, recent developments suggest that there are ways to make it easier. Systems that generate lexical analyzers and parsers already have a long history (e.g. Lex and Yacc), and recent work of Sorin Lerner [23] shows that the same can be done for compiler backends; there is also work suggesting that code generation modules can be automatically generated (e.g. using intermediate languages). Unfortunately, a great number of different compilers are needed in today's software world, and the underlying machine architectures are evolving, as are the languages, so it would be difficult to create verifying compilers for all useful combinations of language and platform, and code verification for such tools still remains very difficult. Major impediments include the unsolvability of discovering loop invariants, the potential unsolvability of loop once they are found, and the further difficulties raised by interactivity, nondeterminism, concurrency, distribution, active agents, and unreliable communication.

A long term approach is to use high level, application specific source languages, in order to greatly simplify source program verification by eliminating many obscure features of current languages. In the meantime, a currently feasible approach is to verify the design of software, instead of its code; experience shows that design verification often leads to better design, and nearly always leads to greater conceptual clarity. An aditional motivation is that the main sources of errors in software are in areas other than code, namely, requirements, specification, and design.

2 The Proof Score Approach

The goal of verification in software engineering is to increase confidence in the correctness of computer-based systems. For software verification to be genuinely useful, careful account must be taken of the context of actual use, including the goals of the system, the underlying hardware, and the capabilities and culture of users. Absolute certainty is not achievable in real applications, due to the uncertainties inherent in implementation and use, e.g., breakdowns in physical infrastructure (such as cables and computers), errors by human operators, and flaws in connected systems (e.g., operating systems and databases).

Fully automatic theorem provers often fail to convey an understanding of their proofs, and they are generally unhelpful when they fail because of user errors in specifications or goals, or due to the lack of a necessary lemma or case split (both of which are very common in practice). It follows that one should seek to make optimal use of the respective abilities of humans and computers, so that computers do the tedious formal calculations, and humans do the high level planning; the tradeoff between detail and comprehension should also be carefully balanced.

Proof scores are a central concept in our approach to meeting these goals; proof scores are instructions to a proof engine, such that when they are executed (or "played"), if everything evaluates as expected, then a desired theorem is proved. Proof scores hide the detailed calculations done by machines, while revealing the proof plan created by humans. Although proof scores can be used to verify code in an imperative language (e.g., as in [20]), it generally makes more sense to verify design rather than code. These techniques differ from model checking [5] in their emphasis on re-usable designs (e.g., for protocols and other algorithms), which may be instantiated in code in many different ways, as well as in their ability to deal with systems that have an infinite number of states, and their natural affinity for abstraction.

We have recognized that many attempts are done to use model checking techniques to prove designs or algorithms which are expressed as, for example, finite state transition systems. But they are usually expressed only using low-level data types and very close to program code. Besides, model checking only gives "yes" or "no with counter example", and does not support interactive analyses or understandings of designs.

Many proof scores have been written in CafeOBJ [6, 9] for verifying properties of distributed systems, especially distributed algorithms, component-based software, and security protocols [8, 27, 31, 32, 34]. Several auxiliary tools have been also been built to support this progress, including PigNose [25], Gateaux, Crème [26], and Kumo [19]. The facilities for behavioral (or observational) proof in CafeOBJ are important for verifying distributed systems, and its advanced module system can be used to express the structure of proofs as well as specifications. In addition, it provides a stable, portable, and reasonably efficient platform for the execution of proof scores, supporting not only term rewriting, but also reasoning in equational logic, rewriting logic, and basic hidden algebra [3]. BOBJ [17] has similar capabilities, featuring a powerful coinduction algorithm that has also been used to verify distributed systems [14].

The following principles can be seen as a requirements analysis for the proof score approach, based on the authors' extensive experience using OBJ [10, 18], BOBJ, and CafeOBJ.

Human Computer Interaction Since fully automatic theorem proving is often infeasible for system verification, it is desirable to integrate the human user in the best possible way: in particular, the proof plans or "scores" should be as readable as possible, and helpful feedback should be provided by machines to humans, in order to maximize their ability to contribute.

Flexible but Clear Structure It is often desirable to arrange the parts of a verification so as to facilitate comprehension. For example, it is often helpful to state and use a result before it is proved, or even (during proof the planning process) before it is known to be true. Long sequences of proof parts can be difficult to understand, especially when there is no obvious immediate connection between two adjacent parts. But such discontinuities are rather common in published proofs, e.g., when a series of lemmas is proved before they are used in proving a main result. This implies that both subgoals and goal/subgoal relations should be very clear.

Flexible Logic Although first order predicate logic is the dominant modern logical system, many other logics are used, and in particular, computer scientists have developed many new logics, e.g., for database systems, knowledge representation, and the semantic web; these include variants of propositional logic, modal logic, intuitionistic logic, higher order logic, and equational logic. It is desirable to be able to support as many of these as possible, as well as their combinations, within a single tool. Moreover, the necessary incompleteness of formal logics that have sufficient expressive power, means that users may want to incorporate new rules of inference "on the fly," for example, to take advantage of a symmetry at the meta level to reduce the cases that need to be considered in a proof.

It is often possible to simulate one logic within another, by imposing a suitable discipline on how its rules are used; in fact, this is precisely what proof scores accomplish. The choice of underlying basic logics for such a purpose is important in at least three dimensions: its efficiency, its simplicity and ease of use, and its ability to support other logics. We believe that equational logic and its variants are the most suitable for this purpose: Equational logics are relatively simple, have many decidable properties with associated efficient algorithms, are relatively easy to read and write, and can support reasoning in most other logics, e.g., by supplying appropriate definitions to an engine that can apply them by as rewrite

rules. By contrast, higher order logic and type theory are much more complex, harder to read and write, harder to mechanize, and harder to reason with, for both humans and machines.

Behavioral Logic Distributed systems consist of abstract machines in which states are hidden, but can be "observed" by certain "attribute" functions with values in basic data types (such as integer or boolean). Behavioral (also called observational) logic is a natural approach to verifying such systems. Three major approaches in this area are: coalgebra (e.g., see the overview [22]); the "observational logic" of Bidoit and Henniker [2, 21]; and hidden algebra [15, 7], on which our own work is based.

CafeOBJ mechanizes the special (but common) case of coinduction in which the effects of methods (or actions) need not be considered for behavioral equivalence, i.e., for which s, s' are behaviorally equivalent if $a(s) = a(s')$ for all applicable attributes a. More sophisticated models can be embedded within this framework, particularly the OTS (Obsevational Transition Systems) model [27] (the power of which is similar to that of unity [4]), and the TOTS (timed OTS) model [28], which provides a logical basis for verifying real time concurrent systems with CafeOBJ.

3 Development of Proof Scores

If a specification is expressed as a set of equations and if the equations can be used as rewriting rules for getting a simplest form of a given expression, then the validity of a statement about the specification can be checked by getting a simplest form of a boolean expression for the statement. The word reduction is used for simplifying an expression by rewriting rules. If a reasoning process can be designed in such a way that all the tedious calculations are done by reductions, and all rules of inference are applied by setting up special contexts for such reductions, and if every reduction gives the expected result, then the desired theorm is proved.

Several well polished small proof scores for data types appeared in OBJ already in 80's, e.g., see [18]. For example, they use reductions to prove induction steps and bases, based on the structure of initial term algebras. Typical examples are proofs of associativity and commutativity of addition for Peano natural numbers, and the identity $n \times (n + 1) = 2 \times (1 + 2 + \cdots + n)$ for any natural number n.

¿From around 1997, the CafeOBJ group at JAIST [3] started to extend the proof score method (1) to apply to distributed and real-time systems, such as classical distributed (and/or real-time) algorithms, component-based software, railway signal systems, secure protocols, etc., (2) to make the method applicable to practical size problems, and (3) to automate the method. As a result, the proof score method using reduction (rewriting) has become a promising way to do serious proofs.

From Static to Dynamic Systems Early proof scores in CafeOBJ included (1) equivalences of functions over natural numbers, (2) equivalences of functions over lists, (3) correctness of simple compilers from expressions to machine code for stack machines, etc. These small proof scores realized an almost ideal combination of high level planning and mechanical reduction. However, even for this class of problems, some non-trivial lemma discovery and/or case spliting is required.

Dynamic systems (i.e. systems with changing states) are common in network/computer based systems, but there is no established methodology in algebraic specification for coping with this class of problems. The CafeOBJ language is designed for writing formal specifications of dynamic systems based on hidden algebra [6, 12, 9]. Many specifications and proof scores for dynamic systems have been done based on hidden algebra semantics, and OTS has been selected as a most promising model. OTS corresponds to a restricted class of hidden algebras, such that it is possible to write specifications for OTS in a fixed standard style that facilitates the development of specifications, and also helps in writing proof scores, since case splits can be suggested by the specifications. The followings publications show stages in the evolution of proof scores for dynamic systems:

- Specifying and verifying mutual exclusion algorithms by incorporating the unity model [27].
- Introduction of a primitive version of OTS [29].
- Introduction of real-time features into OTS/CafeOBJ, and accompanying development of proof scores methodology [28, 11].
- A proper introduction of OTS/CafeOBJ and the related proof score writing method [30, 33].

– Examples of verifications with proof scores in OTS/CafeOBJ [31, 32, 34] (among others).

From Explaining to Doing Proofs A major factor distinguishing the stages of evolution of proof scores is the extent of automation. This is a most important direction of evolution for proof scores, although full automation is not a goal. Automation by reduction is suitable for a mechanical calculation with a focused role and a clear meaning in the context of a larger reasoning process. Early proof scores assisted verification by doing reductions to prove necessary logical statements for a specification. It is intended to gradually codify as many kinds of logical statements as possible into reductions in CafeOBJ. Recently, a fully automatic verification method for a subset of OTS has been developed. This algorithm can be seen as an unification of several techniques for proof scores in OTS/CafeOBJ. The following publications show the stages of automation of proof scores:

– Mainly used for writing formal specifications, but also for proof scores [27].
– Examples with sufficiently complete proof scores [31, 32, 34] (among others).
– A different attempt for automatiing proof scores [25].
– A fully automatic (algorithmic) method of verification for OTS [26][4].

Environment for Proof Score Development The Kumo system [19, 13] provides the proof score approach with greater rigor and better support for proof debugging, although at the cost of learning to use an additional tool. Kumo generates a website, called a proofweb, for each proof attempt that it executes. This website has links to background tutorial material, including tutorial pages for each major proof method used, and for the less familiar mathematical theories used, such as hidden algebra. A somewhat similar experimental tool has been developed for generating and displaying proof scores [36].

4 An Example of Proof Score in CafeOBJ: An Identify-Friend-or-Foe (IFF) Protocol

This section gives a simple but non-trivial example of proof score in CafeOBJ to help readers get a more concrete idea of what proof score is.

Let us consider an Identify-Friend-or-Foe (IFF) protocol[5], which is supposedly used to check if an agent is a member of a group. The IFF protocol can be described as follows:

$$\begin{aligned} &1.\ \mathsf{Check}\ \mathsf{p} \rightarrow \mathsf{q} : \mathsf{r} \\ &2.\ \mathsf{Reply}\ \mathsf{q} \rightarrow \mathsf{p} : \mathcal{E}_{\mathsf{K}}(\mathsf{r}, \mathsf{q}) \end{aligned}$$

We suppose that an agent belongs to only one group, a symmetric key is given to each group, whose members share the key, and keys are different from group to group. If an agent p wants to check if an agent q is a member of the p's group, p generates a fresh random number r and sends it to q as a Check message. On receipt of the message, q replies to the message by sending the random number r received and the ID q, which are encrypted with the symmetric key K of the q's group, to p as a Reply message. On receipt of the message, p tries to decrypt the ciphertext received with the symmetric key of the p's group. If the decryption succeeds and the plaintext consists of r and q, then p concludes that q is a member of the p's group. The protocol is supposed to have the property that if p receives a valid Reply message from q, q is always a member of the p's group. The property is called the IFF property in this paper.

4.1 Modeling and Specification of the Protocol

We suppose that the cryptosystem used is perfect, there is only one legitimate group, all members of the group are trustable, and there are also untrustable agents who are not members. Trustable agents exactly follow the protocol, but untrustable ones may do something against the protocol as well. The combination

[4] This work used the Maude rewriting engine [24] because it provides faster associative and commutative rewriting.

[5] The IFF protocol used in this paper is a modified version of the IFF system described in Subsect. 2.2.2 of [1].

and cooperation of untrustable agents is modeled as the most general intruder (or enemy). The enemy gleans as much information as possible from messages flowing in the network and fakes messages based on the gleaned information, provided that the enemy cannot break the perfect cryptosystem.

We first describe the basic data types used to model the protocol. The visible sorts corresponding to the basic data types and the related operators (if any) are as follows:

- sort `Agent` denotes agents; constant `enemy` denotes the enemy,
- sort `Key` denotes symmetric keys; given an agent `p`, `k(p)` denotes the key of the `p`'s group; operator `p` returns the argument of `k(p)`,
- sort `Rand` denotes random numbers, and
- sort `Cipher` denotes ciphertexts used in the protocol; given a key `k`, a random number `r` and an agent `p`, `enc(k,r,p)` denotes the ciphertext obtained by encrypting `r` and `p` with `k`; operators `k`, `r` and `p` return the first, second and third arguments of `enc(k,r,p)`.

In addition to those visible sorts, the built-in sort `Bool` denoting truth values is used.

The two operators to denote the two kinds of messages are `cm` and `rm`, which are declared as

```
op cm : Agent Agent Agent Rand -> Msg
```

and

```
op rm : Agent Agent Agent Cipher -> Msg,
```

where `Msg` is the visible sort denoting messages. Operators `crt`, `src` and `dst` return the first, second and third arguments of each message; operator `r` returns the fourth argument of a Check message; operator `c` returns the fourth argument of a Reply message. Operators `cm?` and `rm?` check if a given message is a Check message and a Reply message, respectively. The first, second and third arguments of each of `cm` and `rm` mean the actual creator, the seeming sender and the receiver of the corresponding message. The first argument is meta-information that is only available to the outside observer and the agent that has sent the corresponding message, and that cannot be forged by the enemy, while the remaining arguments may be forged by the enemy. The network is modeled as a multiset of messages, which is used as the storage that the intruder can use. The network is also used as each agent's private memory that reminds the agent to send messages, whose first arguments are the agent. Any message that has been sent or put once into the network is supposed to be never deleted from the network. Consequently, the emptiness of the network means that no messages have been sent.

The enemy tries to glean two kinds of values from the network, which are random numbers and ciphertexts. The collections of these values gleaned by the intruder are denoted by operators `rands` and `ciphers`, which are declared as

```
op rands : Network -> ColRands
```

and

```
op ciphers : Network -> ColCiphers,
```

where `Network` is the visible sort denoting networks, `ColRands` is the visible sort denoting collections of random numbers, and `ColCiphers` is the visible sort denoting collections of ciphertexts. The two operators are defined in equations. `ciphers` is defined as follows:

```
eq  C \in ciphers(void) = false .
ceq C \in ciphers(M,NW) = true if rm?(M) and C = c(M) .
ceq C \in ciphers(M,NW) = C \in ciphers(NW) if not(rm?(M) and C = c(M)) .
```

Constant `void` denotes the empty multiset and operator '`,`' in `M,NW` denotes the data constructor of nonempty multisets. The equations say that a ciphertext `C` is available to the enemy iff there exists a Reply message, which includes `C`. `rands` is defined likewise.

We describe the OTS $\mathcal{S}_{\mathtt{IFF}}$ modeling the protocol. The foundation behind the modeling is basically behavioral logic (precisely a restricted class of hidden algebras), but other logics such as the UNITY logic can be incorporated into the proof score approach flexibly and faithfully such as [35] thanks to the flexible logic capability of the CafeOBJ system. Two observations and five parametrized transitions are used. The two observations are denoted by observation operators `nw` and `ur`, which are declared as

```
bop nw : Field -> Network
```

and

```
bop ur : Field -> URands
```

where `Field` is the hidden sort denoting the state space and `URands` is the visible sort denoting sets of random numbers. `nw` and `ur` are used to observe the network and the set of used random numbers. The five transitions are denoted by action operators `sdcm`, `sdrm`, `fkcm1`, `fkrm1` and `fkrm2`, which are declared as follows:

```
bop sdcm : Field Agent Agent Rand -> Field
bop sdrm : Field Agent Msg -> Field
bop fkcm1 : Field Agent Agent Rand -> Field
bop fkrm1 : Field Agent Agent Cipher -> Field
bop fkrm2 : Field Agent Agent Rand -> Field.
```

The first two action operators formalize sending messages exactly following the protocol and the remaining the enemy's faking messages.

The action operators are defined in equations. `sdrm` is defined as follows:

```
ceq nw(sdrm(F,P1,M1)) = rm(P1,P1,src(M1),enc(k(P1),r(M1),P1)) , nw(F)
    if c-sdrm(F,P1,M1) .
eq ur(sdrm(F,P1,M1))  = ur(F) .
ceq sdrm(F,P1,M1)     = F if not c-sdrm(F,P1,M1) .
```

`c-sdrm(F,P1,M1)` equals `M1 \in nw(F) and cm?(M1) and P1 = dst(M1)`, which means that in state `F`, there exists a Check message `M1` in the network, which is addressed to `P1`. If this condition holds, the Reply message denoted by `rm(...)` is put into the network `nw(F)`. `fkrm1` is defined as follows:

```
ceq nw(fkrm1(F,P1,P2,C)) = rm(enemy,P1,P2,C) , nw(F)
    if c-fkrm1(F,P1,P2,C) .
eq ur(fkrm1(F,P1,P2,C))  = ur(F) .
ceq fkrm1(F,P1,P2,C)     = F if not c-fkrm1(F,P1,P2,C) .
```

`c-fkrm1(F,P1,P2,C)` equals `C \in ciphers(nw(F))`, which means that in state `F`, a ciphertext `C` is available to the enemy. The equations say that if a ciphertext `C` is available to the enemy, the enemy can fake and send a Reply message using `C`.

The remaining three action operators are defined as follows:

```
-- for action sdcm
  op c-sdcm : Field Agent Agent Rand -> Bool
  eq c-sdcm(F,P1,P2,R) = not(R \in ur(F)) .
  --
  ceq nw(sdcm(F,P1,P2,R)) = cm(P1,P1,P2,R) , nw(F) if c-sdcm(F,P1,P2,R) .
  ceq ur(sdcm(F,P1,P2,R)) = R ur(F)                if c-sdcm(F,P1,P2,R) .
  ceq sdcm(F,P1,P2,R)     = F                      if not c-sdcm(F,P1,P2,R) .

-- for action fkcm1
  op c-fkcm1 : Field Agent Agent Rand -> Bool
  eq c-fkcm1(F,P1,P2,R) = R \in rands(nw(F)) .
  --
  ceq nw(fkcm1(F,P1,P2,R)) = cm(enemy,P1,P2,R) , nw(F) if c-fkcm1(F,P1,P2,R) .
  eq ur(fkcm1(F,P1,P2,R))  = ur(F) .
  ceq fkcm1(F,P1,P2,R)     = F                         if not c-fkcm1(F,P1,P2,R) .

-- for action fkrm2
  op c-fkrm2 : Field Agent Agent Rand -> Bool
  eq c-fkrm2(F,P1,P2,R) = R \in rands(nw(F)) .
  --
  ceq nw(fkrm2(F,P1,P2,R)) = rm(enemy,P1,P2,enc(k(enemy),R,P1)) , nw(F)
                             if c-fkrm2(F,P1,P2,R) .
```

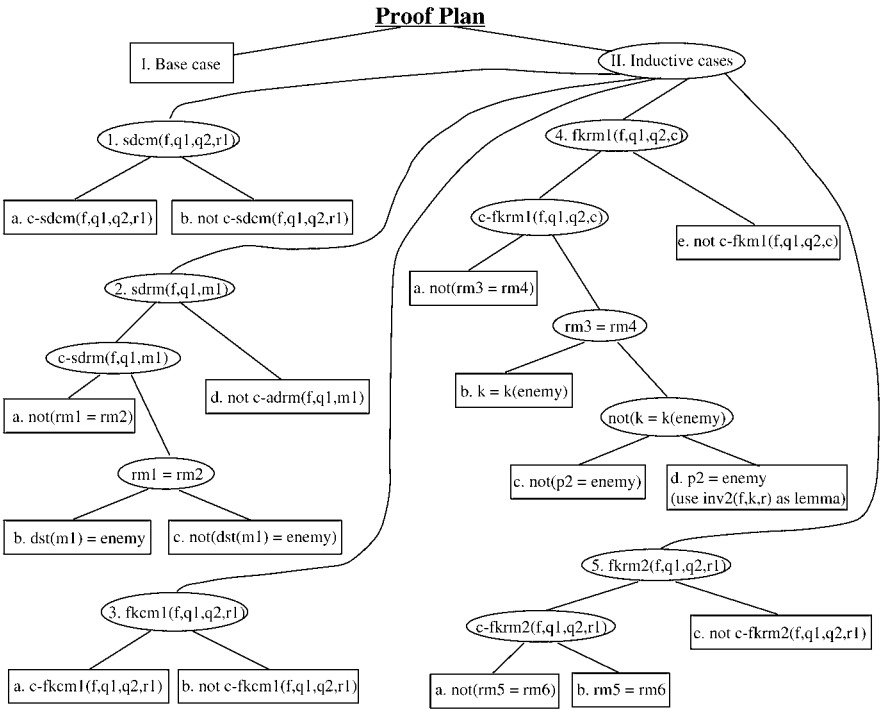

Fig. 1. Proof plan to prove the IFF property

```
eq ur(fkrm2(F,P1,P2,R))  = ur(F) .
ceq fkrm2(F,P1,P2,R)     = F
                           if not c-fkrm2(F,P1,P2,R) .
```

4.2 Proof Score of the IFF Property

The IFF property is denoted by operator `inv1`. `inv1(F,P1,P2,P3,K,R)` equals `(not(K = k(enemy)) and rm(P1,P2,P3,enc(K,R,P2)) \in nw(F)) implies not(P2 = enemy)`. We describe a proof score to prove `inv1(F,P1,P2,P3,K,R)` invariant wrt $\mathcal{S}_{\mathsf{IFF}}$. The proof is done by induction on the number of transitions (or action operators) applied. As described, a proof score is a proof plan to prove that a property holds for a specification. Figure 1 shows the proof plan used to prove `inv1(F,P1,P2,P3,K,R)` invariant wrt $\mathcal{S}_{\mathsf{IFF}}$. In the figure, edges mean case splits, ovals represent intermediate nodes, which mean case splitting in progress, and rectangles represent leaves, which mean results of case splitting. For each rectangle, a fragment of a proof score is written; such a fragment is called a proof passage. For case II.4.d, we use `inv2(f,k,r)` as lemma. `inv2(F,K,R)` equals `enc(K,R,enemy) \in ciphers(nw(F)) implies (K = k(enemy))`.

The proof score to prove `inv1(F,P1,P2,P3,K,R)` invariant wrt $\mathcal{S}_{\mathsf{IFF}}$ corresponds to the proof plan shown in Fig. 1 faithfully, namely that the proof score has clear structure. Therefore, the proof score helps human users comprehend the proof. When writing such a proof score, what you are required to do is mainly case analyses and necessary lemma discoveries. For each case analysis, you write a proof passage for each case obtained by the case analysis. By having the CafeOBJ system execute the proof passage and examining the result returned by the CafeOBJ system, you can judge that the proof is successful in the case, a further case analysis is needed, or a lemma is needed. This is because the proof score approach and the CafeOBJ system provide a flexible human computer interaction mechanism in good balance such that humans make proof plans and machines conduct tedious and detailed computations, and proof scores

have flexible structure. In the proof score approach, lemmas have not necessarily to be proved in advance thanks to flexible structure of proof scores. In the proof of `inv1(F,P1,P2,P3,K,R)`, `inv2(f,k,r)` is used before it is proved.

In this paper, we show the proof passage for case II.4.d, which is as follows:

```
open ISTEP
-- arbitrary objects
  ops q1 q2 : -> Agent .
  op c : -> Cipher .
-- assumptions
  -- eq c-fkrm1(f,q1,q2,c) = true .
  eq enc(k,r,enemy) \in ciphers(nw(f)) = true .
  --
  -- eq rm(enemy,q1,q2,c) = rm(p1,p2,p3,enc(k,r,p2)) .
  eq p1 = enemy .  eq q1 = p2 .  eq q2 = p3 .  eq c = enc(k,r,p2) .
  --
  eq (k = k(enemy)) = false .
  eq p2 = enemy .
-- successor state
  eq f' = fkrm1(f,q1,q2,c) .
-- check if the predicate is true.
  red inv2(f,k,r) implies istep1(p1,p2,p3,k,r) .
close
```

Command `open` makes a temporary module, which imports a given module (`ISTEP` in this case), and command `close` destroys such a temporary module. Necessary operators and equations are declared in `ISTEP`. A comment starts with a double hyphen and terminates at the end of the line. Constants `q1` and `q2` denote arbitrary agents and constant `c` an arbitrary ciphertext. Constants `p1`, `p2` and `p3` denote arbitrary agents, constant `k` an arbitrary key, and constant `r` an arbitrary random number; those constants are declared in `ISTEP` (precisely module `INV` imported by `ISTEP`). Constant `f` denotes an arbitrary state and constant `f'` a successor state of the state; those constants are declared in `ISTEP`. Operator `istep1` denotes a basic formula to prove in each inductive case; `istep1(P1,P2,P3,K,R)` equals `inv1(f,P1,P2,P3,K,R) implies inv1(f',P1,P2,P3,K,R)`. rm3 and rm4 in Fig. 1 correspond to `rm(enemy,q1, q2,c)` and `rm(p1,p2,p3, enc(k,r,p2))` in the proof passage. Instead of `rm(enemy,q1,q2,c) = rm(p1,p2,p3,enc(k,r,p2))`, the four equations `p1 = enemy`, `q1 = p2`, `q2 = p3` and `c = enc(k,r,p2)` are declared because the former can be deduced from the latter with rewriting only, but not vice versa. Instead of `c-fkrm1(f,q1,q2,c) = true`, the equation `enc(k,r,enemy) \in ciphers(nw(f)) = true` is declared because the left-hand side of an equation should be in normal form so as to use the equation effectively as a rewrite rule. Feeding the proof passage into the CafeOBJ system, the CafeOBJ system returns `true` as expected, which means that the proof is successful in this case.

For the remaining cases shown in Fig. 1, similar proof passages are written. The CafeOBJ system also returns `true`'s for those cases. To prove `inv2(F,K,R)` invariant wrt $\mathcal{S}_{\mathsf{IFF}}$, a similar proof score is written, but the proof does not need any lemmas.

5 Conclusions and Future Issues

Promising features and points to be improved of the current proof score method can be summarized as follows:

- Specifications in CafeOBJ can be expressed in relatively high level of abstraction thanks to facilities of user defined data types, OTS (observational transition systems in behavioral logics), and powerful module systems, etc. Proof scores enjoy the same merit and can make it possible to analyze or verify

the specifications or designs in a more higher level of abstraction than other automatic theorem provers or model checkers.
- Proof scores have a high potential for providing a practical new way of doing proofs for designs or specifications, and in particular for providing a relatively low cost approach to compiler verification, and more generally, to verified code produced by application generators.
- Correctness of proof scores is relatively clear but theory for rigid correctness argument needs to be provided.
- Case analyses and lemma discoveries in proof scores can be tedious and difficult, but the current support for systematic management of case splittings and lemma discoveries are not sufficient for large scale and practical problems.

The following are important issues for future research on the proof score method:
- Introducing interactions into the Crème algorithm [26] for guiding high level planning by users while keeping other parts including reductions automatic.
- Farther development of the Kumo/Tatami project (done at UCSD as a subproject of CAFE project) [19] to realize a web (or hypertext) based proof score writing environment.
- Explore institution morphisms [16] as a formal formalization for proof scores.
- Incorporate specific decision procedures like Presburger arithmetic into reductions or as basic units of proof scores other than ordinary reductions.

References

1. Ross Anderson. Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley & Sons, Inc., NY, 2001.
2. Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. Theoretical Computer Science, 165(1):3–55, 1996.
3. CafeOBJ. CafeOBJ web page. `http://www.ldl.jaist.ac.jp/cafeobj/`, 2005.
4. K. Mani Chandy and Jayadev Misra. Parallel Program Design: A Foundation. Addison-Wesley, 1988.
5. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model Checking. MIT Press, Cambridge, MA, 2000.
6. Razvan Diaconescu and Kokichi Futatsugi. CafeOBJ report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing, 6. World Scientific, Singapore, 1998.
7. Razvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. J. UCS, 6(1):74–96, 2000.
8. Razvan Diaconescu, Kokichi Futatsugi, and Shusaku Iida. Component-based algebraic specification and verification in CafeOBJ. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, World Congress on Formal Methods, volume 1709 of Lecture Notes in Computer Science, pages 1644–1663. Springer, 1999.
9. Kokichi Futatsugi. Formal methods in CafeOBJ. In Zhenjiang Hu and Mario Rodríguez-Artalejo, editors, FLOPS, volume 2441 of Lecture Notes in Computer Science, pages 1–20. Springer, 2002.
10. Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 1985 (POPL85), pages 52–66. ACM, 1985.
11. Kokichi Futatsugi and Kazuhiro Ogata. Rewriting can verify distributed real-time systems. In Proc. of International Symposium on Rewriting, Proof, and Computation (PRC2001), pages 60–79. Tohoku Univ., 2001.
12. Joseph Goguen. Hidden algebraic engineering. In Chrystopher Nehaniv and Masami Ito, editors, Algebraic Engineering, pages 17–36. World Scientific, 1998. Papers from a conference at the University of Aizu, Japan, 24–26 March 1997; also UCSD Technical Report CS97–569, December 1997.
13. Joseph Goguen and Kai Lin. Web-based multimedia support for distributed cooperative software engineering. In Proceedings, International Symposium on Multimedia Software Engineering, pages 25–32. IEEE Press, 2000. Papers from a conference Held in Taipei, Taiwan, December 2000.
14. Joseph Goguen and Kai Lin. Behavioral verification of distributed concurrent systems with BOBJ. In Hans-Dieter Ehrich and T.H. Tse, editors, Proceedings of Conference on Quality Software, pages 216–235. IEEE Press, 2003.

15. Joseph Goguen and Grant Malcolm. A hidden agenda. Theoretical Computer Science, 245(1):55–101, 2000.
16. Joseph Goguen and Grigore Roşu. Institution morphisms. Formal Aspects of Computing, 13:274–307, 2002.
17. Joseph Goguen, Grigore Roşu, and Kai Lin. Conditional circular coinductive rewriting. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT'02, volume 2755 of Lecture Notes in Computer Science, pages 216–232. Springer, 2003. Selected papers from a workshop held in Frauenchiemsee, Germany, 24–27 October 2002.
18. Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992.
19. Joseph A. Goguen, Kai Lin, A. Mori, Grigore Rosu, and A. Sato. Distributed cooperative formal methods tools. In Proc. of 1997 International Conference on Automated Software Engineering (ASE '97), November 02-05, 1997, Lake Tahoe, CA, pages 55–62. IEEE, 1997.
20. Joseph A. Goguen and Grant Malcolm. Algebraic Semantics of Imperative Programs. MIT Press, Cambridge, MA, 1996.
21. Rolf Hennicker and Michel Bidoit. Observational logic. In Proc. of Algebraic Methodology and Software Technology (AMAST'98), volume 1548 of Lecture Notes in Computer Science, pages 263–277. Springer, 1999.
22. Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. Bulletin of the European Association for Theoretical Computer Science, 62:222–259, June 1997.
23. Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In PLDI, pages 220–231. ACM, 2003.
24. Maude. Maude web page. `http://maude.cs.uiuc.edu/`, 2005.
25. Akira Mori and Kokichi Futatsugi. CafeOBJ as a tool for behavioral system verification. In Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, ISSS, volume 2609 of Lecture Notes in Computer Science, pages 461–470. Springer, 2002.
26. Masahiro Nakano, Kazuhiro Ogata, Masaki Nakamura, and Kokichi Futatsugi. Automating invariant verification of behavioral specifications. to be published, 2005.
27. Kazuhiro Ogata and Kokichi Futatsugi. Specification and verification of some classical mutual exclusion algorithms with CafeOBJ. In Proceedings of OBJ/CafeOBJ/Maude Workshop at Formal Methods '99, pages 159–177. Theta,, 1999.
28. Kazuhiro Ogata and Kokichi Futatsugi. Modeling and verification of distributed real-time systems based on CafeOBJ. In Proceedings of the 16th International Conference on Automated Software Engineering (16th ASE), pages 185–192. IEEE Computer Society Press, 2001.
29. Kazuhiro Ogata and Kokichi Futatsugi. Specifying and verifying a railroad crossing with CafeOBJ. In Proceedings of the 6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (6th FMPPTA); Part of Proceedings of the 15th IPDPS, page 150. IEEE Computer Society Press, 2001.
30. Kazuhiro Ogata and Kokichi Futatsugi. Rewriting-based verification of authentication protocols. In Proceedings of the 4th International Workshop on Rewriting Logic and its Applications (4th WRLA), ENTCS 71. Elsevier, 2002.
31. Kazuhiro Ogata and Kokichi Futatsugi. Formal analysis of the iKP electronic payment protocols. In Proceedings of the 1st International Symposium on Software Security (ISSS2002), volume 2609 of Lecture Notes in Computer Science, pages 441–460. Springer, 2003.
32. Kazuhiro Ogata and Kokichi Futatsugi. Formal verification of the Horn-Preneel micropayment protocol. In Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (4th VMCAI), volume 2575 of Lecture Notes in Computer Science, pages 238–252. Springer, 2003.
33. Kazuhiro Ogata and Kokichi Futatsugi. Proof scores in the OTS/CafeOBJ method. In Proceedings of the 6th IFIP WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (6th FMOODS), volume 2884 of Lecture Notes in Computer Science, pages 170–184. Springer, 2003.
34. Kazuhiro Ogata and Kokichi Futatsugi. Equational approach to formal verification of SET. In Proceedings of the 4th International Conference on Quality Software (4th QSIC), pages 50–59. IEEE Computer Society Press, 2004.
35. Kazuhiro Ogata and Kokichi Futatsugi. Proof score approach to verification of liveness properties. In 17th International Conference on Software Engineering and Knowledge Engineering (17th SEKE), pages 608–613, 2005.
36. Takahiro Seino, Kazuhiro Ogata, and Kokichi Futatsugi. A toolkit for generating and displaying proof scores in the OTS/CafeOBJ method. In Proceedings of the 6th International Workshop on Rule-Based Programming (RULE'05), Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2005.