CiMPG+F: A Proof Generator & Fixer-upper for CafeOBJ Specifications

Adrián Riesco[1]   Kazuhiro Ogata[2,3]

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

School of Information Science, JAIST, Japan

Research Center for Software Verification, JAIST, Japan
ogata@jaist.ac.jp

ICTAC 2020
Macau S.A.R., China (Online event)

Motivation: CafeOBJ

- CafeOBJ is a language for writing formal specifications and verifying properties of them.
- It implements equational logic by rewriting.
- CafeOBJ specifications are executable, so the specifier can analyze how different terms are reduced.
- In particular, specifiers can write proof scores to prove properties on their specifications.

Motivation: Proof scores

- Proof scores are proof outlines written in CafeOBJ.
- If all proof scores return the expected value when executed (usually `true`), then the corresponding theorems are proved.
- This approach is known as “proving as programming.”

Motivation: Proof scores and CafeInMaude

- An important advantage of this approach is its flexibility: the syntax for performing proofs is the same as for specifying systems.
- However, we lose formality because CafeOBJ does not check proof scores in any way.
- Previously, we presented:
    - An inductive theorem prover (CiMPA).
    - A proof script generator that infers formal proofs from proof scores (CiMPG).
- These tools extended the CafeInMaude compiler, implemented in Maude.
- CafeInMaude takes advantage of Maude metalevel and stores a metarepresentation of proof scores, so we can reason with them at the metalevel.

Motivation: Proof scores and CafeInMaude

- CiMPG had limitations:
  - It generated a formal proof when the proof score was correct and complete.
  - It pointed out the flaws in other case, but it could not suggest how to fix them.
- In this paper we improve the previous version of the tool and present the CafeInMaude Proof Generator and Fixer Upper (CiMPG+F).
- CiMPG+F can:
  - When part of a proof score is missing, infer the rest taking into account the information given by the user.
  - Infer proofs from scratch following a back-tracking algorithm guided by the current goal.

Proving with proof scores

- We present as our running example a simplified cloud synchronization protocol as an observational transition system.
- We have a cloud computer and an arbitrary number of PCs.
- PCs try to keep a value synchronized, so new values appearing in the PCs must be uploaded to the cloud.
- PCs must retrieve new values from the cloud.

Proving with proof scores

- We represent values as natural numbers and consider that larger values are newer to simplify the presentation.
- The cloud computer is represented by:
    - Its status (`statusc`), which takes the values `idlec` and `busy` (a PC is connected).
    - Its current value (`valc`).

Proving with proof scores

- The module `LABELC` defines the labels for the cloud.

```
mod! LABELC {
  [LabelC]
  ops idlec busy : -> LabelC {constr}
  eq (idlec = busy) = false .
}
```

Proving with proof scores

- In turn, the PCs are represented by:
    - Its status (`statusp`), which takes values `idlep`, `gotval` (the PC has fetched the value from the cloud), and `updated` (the PC has updated either its value or the cloud's value).
    - Its current value (`valp`).
    - A temporal value retrieved from the cloud. This value is used to avoid overwriting newer values.

Proving with proof scores

- The module `LABELP` defines the labels for PCs.

```
mod! LABELP {
  [LabelP]
  ops idlep gotval updated : -> LabelP {constr}
  eq (idlep = gotval) = false .
  eq (idlep = updated) = false .
  eq (gotval = updated) = false .
}
```

Proving with proof scores

- The module `VALUE` defines the values for both the cloud and the PCs.

```
mod* VALUE {
  [Value]
  op mv : -> Value {constr}
  op _<=_ : Value Value -> Bool .
  eq (V <= V) = true .
  eq (mv <= V) = true .
}
```

Proving with proof scores

- The module `CLIENT` below requires the existence of a sort `Client`, which stands for PC identifiers:

```
mod* CLIENT {
  [Client]
}
```

Proving with proof scores

- The module `CLOUD` defines the sort `Sys` for the system.
- The values of the system are obtained via *observers*.

```
mod* CLOUD {
  pr(LABELP) pr(LABELC) pr(CLIENT) pr(VALUE)
  [Sys]
  op statusc : Sys -> LabelC
  op valc : Sys -> Value
  op statusp : Sys Client -> LabelP
  ops valp tmp : Sys Client -> Value
```

Proving with proof scores

- The system is built by means of transitions.
- The constructor `init` stands for the initial state.
- The state of both the cloud and PCs is idle and the rest of values are undefined.

```
op init : -> Sys {constr}
eq statusc(init) = idlec .
eq statusp(init,I) = idlep .
```

Proving with proof scores

- The transition `getVal` takes as arguments a system and the identifier of the client that is retrieving the value from the cloud.
- It can only be applied if both the cloud computer and the current computer are idle.
- If these conditions do not hold, then the transition is skipped.

```
op getval : Sys Client -> Sys  {constr}
ceq getval(S,I) = S
 if not (statusp(S,I) = idlep and statusc(S) = idlec) .
```

Proving with proof scores

- If the conditions hold then the status of the cloud is `busy` and the PC goes to `gotVal`:

```
ceq statusc(getval(S,I)) = busy
 if statusp(S,I) = idlep and statusc(S) = idlec .
ceq statusp(getval(S,I),J) = (if I = J
                              then gotval
                              else statusp(S,J)
                              fi)
 if statusp(S,I) = idlep and statusc(S) = idlec .
```

Proving with proof scores

- Similarly, if the conditions hold the temporary value of the computer must be the one of the cloud:

```
ceq tmp(getval(S,I),J) = (if I = J then valc(S) else tmp(S,J) fi)
 if statusp(S,I) = idlep and statusc(S) = idlec .
```

- Because this transition retrieves the value from the cloud and stores it in as temporary value, the current values does not change:

```
eq valc(getval(S,I)) = valc(S) .
eq valp(getval(S,I),J) = valp(S,J) .
```

Proving with proof scores

- The remaining transitions are `gotoidle`, `modval`, and `update`.

```
op gotoidle : Sys Client -> Sys {constr}
op modval : Sys Client -> Sys {constr}
op update : Sys Client -> Sys {constr}
```

Proving with proof scores

- We define seven properties that must hold in the system:

```
ops inv1 inv2 inv6 inv7 : Sys Client -> Bool
ops inv3 inv4 inv5 : Sys Client Client -> Bool
eq inv1(S,I) = (statusp(S,I) = updated implies valp(S,I) = valc(S)) .
eq inv2(S,I) = (statusp(S,I) = gotval implies tmp(S,I) = valc(S)) .
eq inv3(S,I,J) = (statusp(S,I) = updated and
                  statusp(S,J) = gotval implies I = J) .
eq inv4(S,I,J) = (statusp(S,I) = gotval and
                  statusp(S,J) = gotval implies I = J) .
eq inv5(S,I,J) = (statusp(S,I) = updated and
                  statusp(S,J) = updated implies I = J) .
eq inv6(S,I) = not(statusp(S,I) = updated and statusc(S) = idlec) .
eq inv7(S,I) = not(statusp(S,I) = gotval and statusc(S) = idlec) .
```

Proving with proof scores/CiMPA

```
open CLOUD .
 op s : -> Sys .
 ops i j : -> Client .
 eq [inv1 :nonexec] :
     inv1(s,K:Client) = true .
 ...
 eq statusp(s,j) = idlep .
 eq statusc(s) = idlec .
 eq i = j .
 red inv1(s,i) implies
     inv1(getval(s,j),i) .
close
```

```
open CLOUD .
 :goal{
   eq [cloud :nonexec] :
       inv1(S:Sys,C:Client) = true .
   ...}
 :ind on (S:Sys)
 :apply(si)
 :apply(tc)
 :def csb1 = :ctf {
   eq statusp(S#Sys,C#Client) = idlep .}
 :apply(csb1)
 ...
```

Inferring proofs

- For inferring proof scripts for CiMPA, we can use:
  - Simultaneous induction.
  - Theorem of constants.
  - Case splitting (by true/false and terms).
  - Implication with the induction hypotheses, possibly instantiated.
  - Reduction (to check whether the subgoal is reduced to true).
- We need to:
  - Decide when to apply each step.
  - Decide the most appropriate case splitting.
  - Choose and instantiate the induction hypotheses.

Inferring proofs - First steps

- We present the CafeInMaude Proof Generator and Fixer-Upper (CiMPG+F).
- Given a subgoal (possibly the initial one), CiMPG+F uses the possible commands to discharge it.
- First, the variables that require simultaneous induction, if any, are pointed out by the user.
- Induction generates subgoals for each constructor.
- For each subgoal, CiMPG+F applies the theorem of constants when there are variables.
- It is also used for separating different subgoals for the same inductive case.

Inferring proofs - Main algorithm

- CiMPG+F uses a bounded backtracking algorithm.
- It applies case-splitting until the goal is discharged or the bound is reached.

Inferring proofs - Case splitting

- When case splitting is required, the main point is how to decide which one should be applied.
- CiMPG+F chooses the most appropriate terms for case case splitting using the current goal as follows:
  - The goal is reduced.
  - Those non-constructor terms that are not reduced are case-splitting candidates.
  - For each candidate, we check whether the left-hand side of an equation matches it.
  - In that case, the condition that failed is added as candidate (and the previous one is removed).
  - Those candidates that do not match any equation are used for case-splitting.

Inferring proofs - Case splitting

- The kind of case splitting applied to the term depends of the form of the term and its sort.
- CiMPG+F distinguishes between case splitting by true-false and by terms (using constructors).
- It also supports special case splittings for associative sequences.
- It can also infer new candidates by using the induction hypotheses.
- We refer to the full text for the complete algorithm (Section 3.3).

Inferring proofs - Discharging goals

- Goals are discharged when they are reduced to `true`.
- It is usually required to use the induction hypotheses.
- We might have many hypotheses, each of them with free variables that must be instantiated.
- Checking all possible cases would require too much time.
- The user is in charge of introducing a bound in the *number of hypotheses* that can be used at the same time and in the *number of variables* that can be instantiated.
- Even with these limitations the computation might be expensive.

Inferring proofs - Discharging goals

- We use a so-called *pre-instantiation*.
- For pre-instantiating a term, we first reduce $hyp(X_1, ..., X_n) \implies goal$, with X_i variables of the appropriate sort.
- For each equality $c(\cdots, X_i, \cdots) = c(\cdots, t_i, \cdots)$, we instantiate $X_i \mapsto t_i$.
- Pre-instantiations improve the performance up to 300 times faster in the best case.
- We refer to the full text for the complete algorithm (Section 3.2).

Cloud protocol - Case splitting

- We present now how this idea is applied to the Cloud protocol presented before.
- CiMPG+F starts by applying simultaneous induction, which generates five subgoals corresponding to the five constructors (each of them containing the seven properties).
- It then applies the theorem of constants, which generates seven subgoals (one for each property) for the first constructor, `getval`.
- The first subgoal that needs to be proven is `inv1(getval(S,C),i)`.
- The induction hypotheses are `inv1(S, I:Client)`, etc.
- This subgoal is not directly reduced to `true`, so case splitting is required.

Cloud protocol - Case splitting

- The goal is reduced to

```
true xor updated = statusp(getval(S,C),i) and valc(S) = valp(S,i)
                  xor updated = statusp(getval(S,C),i)
```

- Case splitting candidates are initially `updated = statusp(getval(S,C),i)` and `valc(S) = valp(S,i)`.
- For `statusp(getval(S,C),i)` we find an equation that could be applied but the condition `statusp(S,C) = idlep` failed.
- We add `statusp(S,C) = idlep` as candidate and remove `updated = statusp(getval(S,C),i)`.

Cloud protocol - Case splitting

- No terms from the current candidates match a left-hand side, so we generate the case splittings from them.
- For `statusp(S,C) = idlep` it generates a case splitting by terms for `statusp(S,C)`.
- For `valc(S) = valp(S,i)` it generates a case splitting by equations, as well as case splitting by terms for `valc(S)` and `valp(S,i)`.
- In fact, the proof for this goal starts with `statusp(S,C)`.

Cloud protocol - Discharging goals

- After three case splittings, the goal can be reduced to `true`.
- It is required to use an implication with the induction hypothesis.
- Given the goal `inv1(getval(S,C),i)` and the hypothesis `inv1(S, I:Client)`, CiMPG+F uses the substitution `I : Client` ↦ `i`.

Understanding failures

- In case a proof is not found, CiMPG+F shows:
    - The case splittings that were applied.
    - The subgoals that could be discharged.
    - The subgoales that failed because
        1. No more case splittings were available or
        2. The bound on the depth of the backtracking algorithm is reached.
- For each goal and subgoal it shows the case splitting that was tried and how it behaved.
- Because several case splittings are in general possible for each goal it shows all of them, numbering them and using indentation for the sake of readability.

Cloud - Understanding failures

```
*** Goal 1, Try 1 -  inv1(getval(S#Sys,C#Client),i@Client) - Failure
:def csb1 = :ctf {eq valc(S#Sys) = valp(S#Sys,i@Client).}
:apply(csb1)

 *** Goal 1-1 Success by reduction
 :apply (rd)

 *** Goal 1-2 cannot be discharged. Maximum depth reached.

*** Goal 1, Try 2 -  inv1(getval(S#Sys,C#Client),i@Client) - Failure
 :def csb1 = :ctf [statusc(S#Sys) .]
 :apply(csb1)

 *** Goal 1-1 Success by implication and reduction.
 :imp [proofCLOUD] by {i:Client <- i@Client ;}
 :apply (rd)

 *** Goal 1-2 cannot be discharged. Maximum depth reached.
```

Guiding proofs with proof scores

- For large specifications each step is expensive because many case splittings are possible and many implications and instantiations can be used.
- Fully automated theorem provers cannot deal with large or complex specifications, so user interaction is required.
- It might be the case that the user is stuck in a particular subgoal; it might help the user to focus on more promising case splittings.
- It is possible to feed CiMPG+F with an incomplete proof score, which must contain the case splittings that will be used first.
- The CiMPG algorithm will reconstruct the proof to that point and then execute CiMPG+F to try to generate the rest of the proof.

Guiding proofs with proof scores

- Note that a single equation might not determine the case splitting.
- CiMPG+F collects the equations from the open-close environments related to each inductive case and analyzes whether:
  - All the equations required to the splitting are present.
  - In this case the case splitting is applied and the standard analysis used by CiMPG continues.
  - If the splitting is not unambiguously identified, we need to try each of them and, if the proof fails, try again with the next possible splitting.
  - In this case the CiMPG+F algorithm is required, but the non-determinism is reduced.

Benchmarks

- The benchmarks performed thus far give us confidence in its applicability.

Name	Spec. size	Proof size	Time	Description
2p-mutex	58	23	<1s	2 processes mutex
ABP	320	1370	∼2.5h	Alternating Bit Protocol
Cloud	127	530	<1s	Simplified cloud synch. protocol
NSLPK	188	342	617s	Authentication protocol NSLPK
Qlock	124	88	<1s	Variant of Dijkstra's semaphore
SCP	182	200	340s	Simple Communication Protocol
TAS	73	88	<1s	Mutual exclusion protocol

Concluding remarks

- We have presented CiMPG+F, a tool that tries to automatically prove properties of CafeOBJ specifications.
- These proofs can be based on known information.
- They can be also completely generated by using a bounded depth-first search directed by the current goal and system.
- CiMPG+F optimizes some subtasks, like instantiation of free variables, and provides three parameters to customize the generated proofs.
- The performance of the tool has been tested with some benchmarks.

Ongoing work

- We are interested in informing the user whether a goal is not provable, which happens when we are able to reduce it to `false` and there are no contradictions in the module.
- We do not want to limit its application to the generation of complete proofs.
- It is worth making CiMPG+F more interactive.
- It could graphically show information about:
  - Those branches that have been traversed without reaching a result.
  - Those branches that have not been traversed yet.
- This interaction would ease the proofs of advanced protocols, which cannot be automatically generated in general.

Thanks!

- Please visit the webpage for more details: `https://github.com/ariesco/CafeInMaude`
- Questions are welcome! (`ariesco@ucm.es`)