~~Model Checking: A Remedy for State Explosion~~

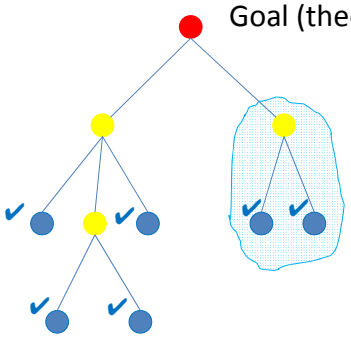
Proof Script Generation from Proof Scores (2)

Kazuhiro Ogata (JAIST, Japan)

i628e information processing theory

Motivation

Proof tree A proof can be regarded as a tree.



An instance of a proof rule

$$\frac{B \quad C}{A}$$

For example,

$$\frac{p(0) \quad (\forall x\colon Nat)\,\big(p(x) \Rightarrow p(x+1)\big)}{(\forall x\colon Nat)\,p(x)}$$

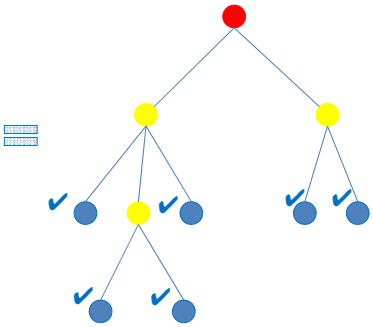
Motivation

Proof assistant

A sequence of commands (or tactics), or a *proof script*

Pros : it guarantees that once a proof is done, it is correct.

Pros : less flexible



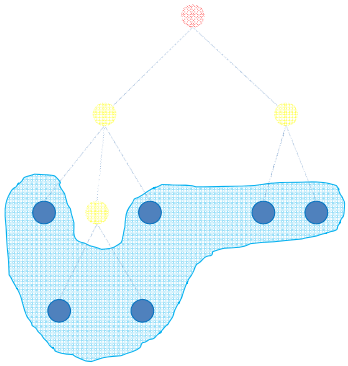
Motivation

Proof scores Proofs (or proof plans) written in an alg. spec. lang.

Pros : flexible, can be written as programs, etc.

Cons : subject to human errors

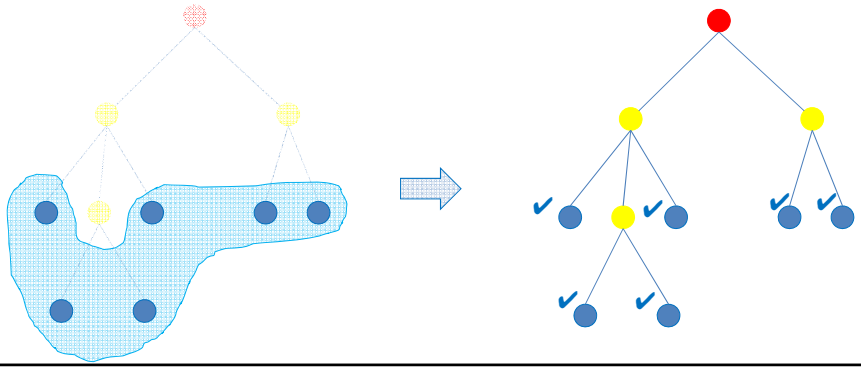
Only consisting of the leaf nodes of a proof tree



Motivation

Would like to check if proof scores are correct

One possible way to do so is to construct (or infer) proof scripts from proof scores



Outline

1. Review of lecture 1
2. CiMPA – CafeInMaude Proof Assistant
3. CiMPG – CafeInMaude Proof Generator
4. Summary & future directions

Review of lecture 1

A mutual exclusion protocol called Qlock, an abstract version of the Dijkstra binary semaphore.

Pseudo code for process p:

Loop "Remainder Section"
rs: enqueue(*queue*,*p*);
ws: repeat until top(*queue*) = *p*;
"Critical Section"
cs: dequeue(*queue*);

Each process p is located at rs, ws or cs.

Initially, p is located at rs & *queue* is empty.

Review of lecture 1

Treating states as black boxes but possible to observe values from them. Given a state s & a process ID i,

$\mathrm{queue}(s)$ the content of *queue* used in Qlock

$\mathrm{pc}(s, i)$ the location where process i is located

queue & pc are called *observer functions*.

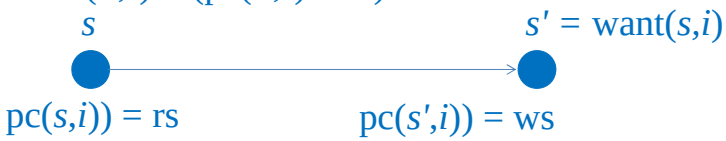
Let init be an arbitrary initial state.

eq $\mathrm{queue}(\mathrm{init}) = \mathrm{empty}$.
eq $\mathrm{pc}(\mathrm{init}, I) = \mathrm{rs}$.

Review of lecture 1

```
ceq pc(want(S,I),J)
  = (if I = J then ws else pc(S,J) fi)        if c-want(S,I) .
ceq queue(want(S,I)) = enq(queue(S),I)        if c-want(S,I) .
ceq want(S,I)        = S                      if not c-want(S,I) .
```

where $c\text{-want}(S,I) = (pc(S,I) = rs)$



$pc(s',j) = pc(s,j)$ **if** $i \neq j$
$queue(s') = enq(queue(s),i)$

Loop "Remainder Section"
rs: enqueue(*queue*);
ws: repeat until top(*queue*) = *p*;
"Critical Section"
cs: dequeue(*queue*);

want is called a *transition functions*.

Review of lecture 1

```
ceq pc(try(S,I),J)
= (if I = J then cs else pc(S,J) fi)    if c-try(S,I) .
eq  queue(try(S,I)) = queue(S) .
ceq try(S,I)        = S                 if not c-try(S,I) .
```

where $c\text{-try}(S,I) = (pc(S,I) = ws)$ and $top(queue(S)) = I)$

$pc(s',j) = pc(s,j)$ **if** $i \neq j$
$queue(s') = queue(s)$

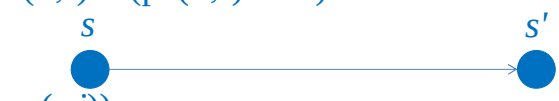
Loop "Remainder Section"
rs: enqueue(*queue*);
ws: repeat until top(*queue*) = *p*;
"Critical Section"
cs: dequeue(*queue*);

try is called a *transition functions*.

Review of lecture 1

```
ceq pc(exit(S,I),J)
  = (if I = J then rs else pc(S,J) fi)      if c-exit(S,I) .
ceq queue(exit(S,I)) = deq(queue(S))        if c-exit(S,I) .
ceq exit(S,I)        = S                    if not c-exit(S,I) .
```

where c-exit(S,I) = (pc(S,I) = cs)



$pc(s',j) = pc(s,j)$ **if** $i \neq j$
$queue(s') = deq(queue(s))$

```
Loop "Remainder Section"
  rs: enqueue(queue);
  ws: repeat until top(queue) = p;
      "Critical Section"
  cs: dequeue(queue);
```

exit is called a *transition functions*.

Review of lecture 1

Mutual exclusion property: there is always at most one process in the critical section.

```
eq inv1(S:Sys,I:Pid,J:Pid)
  = (((pc(S,I) = cs) and pc(S,J) = cs) implies I = J) .
```

To prove $(\forall s : \mathrm{Sys})(\forall \mathrm{i,j: Pid})\ \mathrm{inv1}(s,\mathrm{i},\mathrm{j})$, we need

```
eq inv2(S:Sys,I:Pid) = (pc(S,I) = cs implies top(queue(S)) = I) .
```

$(\forall s : \mathrm{Sys})(\forall \mathrm{i,j: Pid})\ \mathrm{inv1}(s,i,j)$ &
$(\forall s : \mathrm{Sys})(\forall \mathrm{i: Pid})\ \mathrm{inv2}(s,i)$ are proved by simultaneous (structural) induction on s.

Review of lecture 1

✔

```
open QLOCK .
ops i j : -> Sys .
red inv1(init,i,j) .
close
```

Base case for inv1

Induction case in which want is taken into account for inv1

```
open QLOCK .
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq (pc(s,k) = rs) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

✔

Review of lecture 1

```
open QLOCK .
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq pc(s,k) = rs .  eq i = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

✔

```
open QLOCK .
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq pc(s,k) = rs .  eq (i = k) = false .  eq j = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

✔

Review of lecture 1

```
open QLOCK .
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq pc(s,k) = rs .  eq (i = k) = false .  eq (j = k) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

✔

CiMPA – CafeInMaude Proof Assistant

```
:goal{
eq [inv1 :nonexec] : inv2(S:Sys,P:Pid) = true .
eq [inv11 :nonexec] : inv1(S:Sys,P:Pid,P0:Pid) = true .    }
:ind on (S:Sys)
:apply(si)
…
:apply(tc)
:apply (rd)
:apply (rd)
…
```

CiMPA – CafeInMaude Proof Assistant

```
:apply(tc)
:def csb17 = :ctf {eq pc(S#Sys,P#Pid) = rs .}
:apply(csb17)
:def csb18 = :ctf {eq P@Pid  = P#Pid .}
:apply(csb18)
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb19 = :ctf {eq P0@Pid  = P#Pid .}
:apply(csb19)
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
```


CiMPA – CafeInMaude Proof Assistant

```
:apply (rd)
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb20 = :ctf {eq pc(S#Sys,P#Pid) = rs .}
:apply(csb20)
:def csb21 = :ctf {eq P@Pid  = P#Pid .}
:apply(csb21)
:imp [inv1] by {P:Pid <- P@Pid ;}
:apply (rd)
```

CiMPA – CafeInMaude Proof Assistant

```
:def csb22 = :ctf [queue(S#Sys) .]
:apply(csb22)
:imp [inv1] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [inv1] by {P:Pid <- P@Pid ;}
:apply (rd)
:imp [inv1] by {P:Pid <- P@Pid ;}
:apply (rd)
```


CiMPA – CafeInMaude Proof Assistant

```
:goal{
eq [inv1 :nonexec] : inv2(S:Sys,P:Pid) = true .
eq [inv11 :nonexec] : inv1(S:Sys,P:Pid,P0:Pid) = true .    }
:ind on (S:Sys)
:apply(si)
```

$(\forall s : \mathrm{Sys})(\forall \mathrm{i, j: Pid})\ \mathrm{inv1}(s, i, j)$ &
$(\forall s : \mathrm{Sys})(\forall \mathrm{i: Pid})\ \mathrm{inv2}(s, i)$ are proved by simultaneous (structural) induction on s.

CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .
red inv1(init,P:Pid,P0:Pid) .
red inv2(init,P:Pid) .
close
```

```
:apply(tc)
```

```
open QLOCK .
ops P@Pid P0@Pid : -> Pid .
red inv1(init,P@Pid,P0@Pid) .
close
```

```
open QLOCK .
ops P@Pid P0@Pid : -> Pid .
red inv2(init,P@Pid) .
close
```

```
:apply(rd)
:apply(rd)
```


CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .
op S#Sys : -> Sys .    ops P@Pid P0@Pid P#Pid : -> Pid .
eq [inv1 :nonexec] : inv2(S#Pid,P:Pid) = true .
eq [inv11 :nonexec] : inv1(S:Sys,P:Pid,P0:Pid) = true .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:def csb17 = :ctf {eq pc(S#Sys,P#Pid) = rs .}
:apply(csb17)
```

```
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
open QLOCK .    …
eq (pc(S#Sys,P#Pid) = rs) = false .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:def csb18 = :ctf {eq P@Pid  = P#Pid .}
:apply(csb18)
```

```
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs .
eq P@Pid  = P#Pid .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs .
eq (P@Pid  = P#Pid) = false .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```


CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs .
eq P@Pid  = P#Pid .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
```

```
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs .
eq P@Pid  = P#Pid .
red inv1(S#Sys,P@Pid,P0@Pid) implies
    inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:apply (rd)
```

CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .  …
eq pc(S#Sys,P#Pid) = rs . eq (P@Pid  = P#Pid) = false .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:def csb19 = :ctf {eq P0@Pid  = P#Pid .}
:apply(csb19)
```

```
open QLOCK .  …
eq pc(S#Sys,P#Pid) = rs . eq (P@Pid  = P#Pid) = false .
eq P0@Pid  = P#Pid .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
open QLOCK .  …
eq pc(S#Sys,P#Pid) = rs . eq (P@Pid  = P#Pid) = false .
eq (P0@Pid  = P#Pid) = false .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```


CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .  …
eq pc(S#Sys,P#Pid) = rs . eq (P@Pid  = P#Pid) = false .
eq P0@Pid  = P#Pid .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
```

```
open QLOCK .  …
eq pc(S#Sys,P#Pid) = rs . eq (P@Pid  = P#Pid) = false .
eq P0@Pid  = P#Pid .
red inv1(S#Sys,P@Pid,P0@Pid) implies
    inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:apply (rd)
```

CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs . eq (P@Pid  = P#Pid) = false .
eq (P0@Pid  = P#Pid) = false .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
```

```
open QLOCK .    …
eq pc(S#Sys,P#Pid) = rs . eq (P@Pid  = P#Pid) = false .
eq (P0@Pid  = P#Pid) = false .
red inv1(S#Sys,P@Pid,P0@Pid) implies
     inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:apply (rd)
```


CiMPA – CafeInMaude Proof Assistant

```
open QLOCK .    …
eq (pc(S#Sys,P#Pid) = rs) = false .
red inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
```

```
open QLOCK .    …
eq (pc(S#Sys,P#Pid) = rs) = false .
red inv1(S#Sys,P@Pid,P0@Pid) implies
     inv1(want(S#Sys,P#Pid),P@Pid,P0@Pid) .
close
```

```
:apply (rd)
```

CiMPA – CafeInMaude Proof Assistant

If some commands (or tactics) in a proof script are missing, CiMPA complains.

Therefore, CiMPA can avoid human errors.

CiMPG – CafeInMaude Proof Generator

Input: proof score

```
open QLOCK .
ops i j : -> Sys .
red inv1(init,i,j) .
close
 ...
open QLOCK .
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq (pc(s,k) = rs) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
 ...
```

CiMPG

Output: proof script

```
:goal{
eq [inv1 :nonexec] : inv2(S:Sys,P:Pid) = true .
eq [inv11 :nonexec] : inv1(S:Sys,P:Pid,P0:Pid) = true .    }
:ind on (S:Sys)
:apply(si)
…
:apply(tc)
:def csb17 = :ctf {eq pc(S#Sys,P#Pid) = rs .}
:apply(csb17)
:def csb18 = :ctf {eq P@Pid  = P#Pid .}
:apply(csb18)
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
:apply (rd)
:def csb19 = :ctf {eq P0@Pid  = P#Pid .}
:apply(csb19)
:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}
…
```

CiMPG – CafeInMaude Proof Generator

Necessary to have the directive :id(inv1) in each open-close fragment and one special open-close fragment that has the directive :proof(inv1).

```
open QLOCK .
:id(inv1)
ops i j : -> Sys .
red inv1(init,i,j) .
close

        ...
```

```
open QLOCK .
:id(inv1)
op s : -> Sys .  ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq (pc(s,k) = rs) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close

        ...
```

```
open QLOCK .
  :proof(inv1)
close
```


CiMPG – CafeInMaude Proof Generator

Looking at all open-close fragments in which :id(inv1) is written and one special open-close fragment in which :proof(inv1) is written, CiMPG recognizes what to prove.

More concretely, it looks at terms to reduce except for implications:

```
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
red inv2(s,j) implies inv1(s,i,j) implies inv1(try(s,k),i,j) .
red inv1(s,i,k) implies inv2(s,i) implies inv2(exit(s,k),i) .
```

Then, it generates the following:

```
:goal{
eq [inv1 :nonexec] : inv2(S:Sys,P:Pid) = true .
eq [inv11 :nonexec] : inv1(S:Sys,P:Pid,P0:Pid) = true .    }
:ind on (S:Sys)
:apply(si)
```

CiMPG – CafeInMaude Proof Generator

For each constructor of Sys, such as wait, and each state predicate to tackle, such as inv1, it looks at all open-close fragments in which wait is considered.

```
open QLOCK .
op s : -> Sys .
ops i j k : -> Pid .
eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
eq pc(s,k) = rs .  eq (i = k) = false .  eq j = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

It looks at the equations used for case splitting.

CiMPG – CafeInMaude Proof Generator

From the four open-close fragments

```
open QLOCK .  …
eq pc(s,k) = rs .  eq i = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

```
open QLOCK .  …
eq pc(s,k) = rs .  eq (i = k) = false .
eq j = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

```
open QLOCK .  …
eq pc(s,k) = rs .  eq (i = k) = false .
eq (j = k) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

```
open QLOCK .  …
eq (pc(s,k) = rs) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

CiMPG – CafeInMaude Proof Generator

it generates the following:

```
:def csb17 = :ctf {eq pc(S#Sys,P#Pid) = rs .}
:apply(csb17)
```

From the three open-close fragments

```
open QLOCK .  …
eq pc(s,k) = rs .  eq i = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

```
open QLOCK .  …
eq pc(s,k) = rs .  eq (i = k) = false .
eq (j = k) = false .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

```
open QLOCK .  …
eq pc(s,k) = rs .  eq (i = k) = false .
eq j = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```


CiMPG – CafeInMaude Proof Generator

it generates the following:

```
:def csb18 = :ctf {eq P@Pid  = P#Pid .}
:apply(csb18)
```

From the one open-close fragment

```
open QLOCK .  …
eq pc(s,k) = rs .  eq i = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

because the two equations used for case splitting have been already generated

CiMPG – CafeInMaude Proof Generator

it generates the following:

:imp [inv11] by {P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;}

From the one open-close fragment

```
open QLOCK .    …
eq pc(s,k) = rs .  eq i = k .
red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

because this open-close fragment has been already generated

it generates the following:

:apply (rd)

CiMPG – CafeInMaude Proof Generator

If some open-close fragments are missing, CiMPG informs users that there are some missing open-close fragments.

If CiMPG successfully generates a proof script from a proof score, then the proof score is correct because the proof script is checked by CiMPA at the same time.

CiMPG – CafeInMaude Proof Generator

Table 1. Case Studies Used to Test the Tool

Name	LOC	Commands	Time
2p-mutex	50 + 70	28	724 ms
TAS	50 + 230	76	9762 ms
QLOCK	100 + 400	112	23297 ms
SCP	182 + 664	202	61989 ms
NSLPK	180 + 1100	284	118207 ms
Cloud	120 + 1700	383	132118 ms
ABP	264 + 8655	1329	5 h 13 m

Why it took much time for ABP is because associativity is used.

$$(x \mathbin{@} y) \mathbin{@} z = x \mathbin{@} (y \mathbin{@} z)$$

Summary and future directions

Summary

Qlock has been used as an example to describe CiMPA and CiMPG.

Publications

For CiMPA & CiMPG,

Adrián Riesco, Kazuhiro Ogata: Prove it! Inferring Formal Proof Scripts from CafeOBJ Proof Scores. ACM Trans. Softw. Eng. Methodol. 27(2): 6:1-6:32 (2018)

For CafeInMaude,

Adrián Riesco, Kazuhiro Ogata, Kokichi Futatsugi: A Maude environment for CafeOBJ. Formal Asp. Comput. 29(2): 309-334 (2017)

Summary and future directions

Future directions

Extending CiMPG so that it can fix incomplete proof scores given

Extending CiMPG so that it can construct proof scripts even if any proof scores are not given

Integrating some human interactions into an extension of CiMPG

Integrating (semi-)automatic lemma conjecture mechanism into an extension of CiMPG