Modeling, Specification, and Verification of QLOCK in OTS/CafeOBJ

Lecture Note 06+07
Formal Methods (i613-0912)
Topics

- What is QLOCK?
- Modeling and Description of QLOCK in OTS
- Formal specification of QLOCK in CafeOBJ
- Formal specification of mutual exclusion property of QLOCK
- Systematic construction of Proof Score for verifying the mutual exclusion property
Modeling, Specifying, and Verifying (MSV) in CafeOBJ

1. By understanding a problem to be modeled/specified, determine several sorts of objects (entities, data, agents, states) and operations (functions, actions, events) over them for describing the problem.

2. Define the meanings/functions of the operations by declaring equations over expressions/terms composed of the operations.

3. Write proof scores for properties to be verified.
MSV with proof scores in CafeOBJ

Understand problem and construct model

Write system spec SPsys and Write property spec SPprop

Construct proof score of SPprop w.r.t. SPsys
An example: mutual exclusion protocol

Assume that many agents (or processes) are competing for a common equipment, but at any moment of time only one agent can use the equipment. That is, the agents are mutually excluded in using the equipment. A protocol (mechanism or algorithm) which can achieve the mutual exclusion is called “mutual exclusion protocol”. 
QLOCK (locking with queue): a mutual exclusion protocol

Each agent $i$ is executing:

- **Put its name $i$ into the bottom of the queue**

- **Remainder Section**

- **Remove/get the top of the queue**

- **Critical Section**

: atomic action

- **Is $i$ at the top of the queue?**
  - **true**
  - **false**

- **wt**
- **rm**
- **cs**
QLOCK: basic assumptions/characteristics

- There is only one queue and all agents/processes share the queue.
- Any basic action on the queue is inseparable (or atomic). That is, when any action is executed on the queue, no other action can be executed until the current action is finished.
- There may be unbounded number of agents.
- In the initial state, every agents are in the remainder section (or at the label $\text{rm}$), and the queue is empty.

The property to be shown is that at most one agent is in the critical section (or at the label $\text{cs}$) at any moment.
Global (or macro) view of QLOCK
Modeling QLOCK (via Signature Diagram) with OTS (Observational Transition System)
Schematic signature diagram for OTS

Visible Sorts (Data) → Action (method) → Hidden Sort (State Space) → Observation (attribute) → Visible Sorts (Data)

Action (method) → Hidden Sort (State Space) → Observation (attribute) → Visible Sorts (Data)

Visible Sorts (Data) → Coherent → Visible Sorts (Data)

VSs

HSs
Signature for QLOCKwithOTS

- **Sys** is the sort for representing the state space of the system.
- **Pid** is the sort for the set of agent/process names.
- **Label** is the sort for the set of labels; i.e. \{rm, wt, cs\}.
- **Queue** is the sort for the queues of **Pid**
- **pc** (program counter) is an observer returning a label where each agent resides.
- **queue** is an observer returning the current value of the waiting queue of **Pid**.
- **want** is an action for agent \(i\) of putting its name/id into the queue.
- **try** is an action for agent \(i\) of checking whether its name/id is at the top of the queue.
- **exit** is an action for agent \(i\) of removing/getting its name/id from the top of the queue.
CafeOBJ signature for QLOCKwithOTS

-- state space of the system
* [Sys] *

-- visible sorts for observation
[Queue Pid Label]

-- observations
bop pc : Sys Pid -> Label
bop queue : Sys -> Queue

-- any initial state
init : -> Sys (constr)

-- actions
bop want : Sys Pid -> Sys (constr)
bop try : Sys Pid -> Sys (constr)
bop exit : Sys Pid -> Sys (constr)
Module LABEL specifying (via tight denotation) “labels”

mod! LABEL {
    -- EQL is a meta module
    -- for making _=_ predicate available
    inc(EQL)

    [LabelConst < Label]
    ops rm wt cs : -> LabelConst {constr}
    vars Lc1 Lc2 : LabelConst
    eq (Lc1 = Lc2) = (Lc1 == Lc2) .
}

Predicate (_ = _) defines identity relation among rm, wt, and cs.
Module PID specifying (via loose denotation) “agent/process names/identifiers”

```plaintext
mod* PID {
  inc(EQL)
  [PidConst < Pid < PidErr]
  vars Pc1 Pc2 : PidConst
  eq (Pc1 = Pc2) = (Pc1 == Pc2) .
}
```

- Equality of element of PidConst is judged by (_ == _) .
Module QUEUE specifying “queue” (1) -- a parameterized module

mod* TRIV= {  
  inc (EQL)  
  [EltConst < Elt < EltErr]  
}

mod* QUEUE (D :: TRIV=) {  
  inc (EQL)  

  -- sort Queue and its constructors  
  [Queue < QueueErr]  
  op empty : -> Queue {constr}  
  op _,_ : Queue Elt.D -> Queue {constr}  

  -- equality over Queue  
  eq (empty = empty) = true .  
  eq (Q1:Queue,E1:Elt = Q2:Queue,E2:Elt)  
    = (Q1 = Q2) and (E1 = E2) .  

...
Module QUEUE specifying “queue” (2) -- a parameterized module

```
op put : EltErr.D QueueErr -> QueueErr
op put : Elt.D Queue -> Queue
eq put(X,empty) = empty,X .
eq put(X,(Q,Y)) = put(X,Q),Y .
op get : QueueErr -> QueueErr
eq get((Q,X)) = Q .
eq (get(empty) = Q:Queue) = false .
 Error declaration

eq top((Q,X)) = X .
eq (top(empty) = E:Elt.D) = false .
 Error declaration

eq empty?(empty) = true .
eq empty?((Q,X)) = false . }
```
QLOCK using operators in the CafeOBJ module QUEUE

Each agent \( i \) is executing:

- **Critical Section**
  - \( \text{top}(queue) = i \)
  - \( \text{cs} \)
  - \( \text{try} \)
  - \( \text{false} \)

- **Remainder Section**
  - \( \text{put}(queue, i) \)
  - \( \text{rm} \)
  - \( \text{want} \)
  - \( \text{wt} \)
  - \( \text{get}(queue) \)
  - \( \text{exit} \)
  - \( \text{true} \)
Module QLOCK specifying "QLOCK" (1)

```plaintext
mod* QLOCK {
    inc(LABEL)
    inc(QUEUE(PID{sort Elt -> Pid, sort EltConst -> PidConst, sort EltErr -> PidErr}))

    -- system space of QLOCK
    *[Sys]*
    -- initial state of QLOCK
    op init : -> Sys {constr}

    -- actions
    bop want : Sys Pid -> Sys {constr}
    bop try  : Sys Pid -> Sys {constr}
    bop exit : Sys Pid -> Sys {constr}

    -- observations
    bop pc    : Sys Pid -> Label
    bop queue : Sys -> Queue
}
```
Module QLOCK specifying “QLOCK” (2)

-- initial state
eq \text{pc}(\text{init}, I: \text{Pid}) = \text{rm} .
eq \text{queue}(\text{init}) = \text{empty} .

var S : \text{Sys} . \text{vars} I J : \text{Pid} .
-- want
op c\text{-}want : \text{Sys} \text{ Pid} \rightarrow \text{Bool} \{\text{strat: (0 1 2)}\}
eq c\text{-}want(S, I) = (\text{pc}(S, I) = \text{rm}) .

--
ceq \text{pc}(\text{want}(S, I), J)
  = (\text{if } I = J \text{ then } \text{wt} \text{ else } \text{pc}(S, J) \text{ fi})
  \text{if } c\text{-}want(S, I) .
ceq \text{queue}(\text{want}(S, I)) = \text{put}(I, \text{queue}(S))
  \text{if } c\text{-}want(S, I) .
ceq \text{want}(S, I) = S
  \text{if } \text{not } c\text{-}want(S, I) .
Module QLOCK specifying “QLOCK” (3)

```plaintext
-- try
  op c-try : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-try(S,I) = (pc(S,I) = wt and top(queue(S)) = I).
  --
  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).

-- exit
  op c-exit : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-exit(S,I) = (pc(S,I) = cs).
  --
  ceq pc(exit(S,I),J)
      = (if I = J then rm else pc(S,J) fi) if c-exit(S,I).
  ceq queue(exit(S,I)) = get(queue(S)) if c-exit(S,I).
  ceq exit(S,I) = S if not c-exit(S,I).
```

(_ =*= _) is congruent for OTS

The binary relation \((S1:Sys =*= S2:Sys)\) is defined to be true iff \(S1\) and \(S2\) have the same observation values.

OTS style of defining the possible changes of the values of observations is characterized by the equations of the form:
\[
o(a(s,d),d') = \ldots o_1(s,d_1) \ldots o_2(s,d_2) \ldots o_n(s,d_n) \ldots
\]
for appropriate data values of \(d,d',d_1,d_2,\ldots,d_n\).

It can be shown that OTS style guarantees that \((_ =*= _)\) is congruent with respect to all actions.
$R_{\text{QLOCK}}$ (set of reachable states) of $\text{OTS}_{\text{QLOCK}}$ (OTS defined by the module QLOCK)

**Signature determining $R_{\text{QLOCK}} = \mathbf{Sys}$**

-- initial state

\[
\text{op init} : \rightarrow \mathbf{Sys} \{\text{constr}\}
\]

-- actions

\[
\begin{align*}
\text{bop want} : \mathbf{Sys} \times \text{Pid} & \rightarrow \mathbf{Sys} \{\text{constr}\} \\
\text{bop try} : \mathbf{Sys} \times \text{Pid} & \rightarrow \mathbf{Sys} \{\text{constr}\} \\
\text{bop exit} : \mathbf{Sys} \times \text{Pid} & \rightarrow \mathbf{Sys} \{\text{constr}\}
\end{align*}
\]

**Recursive definition of $R_{\text{QLOCK}} = \mathbf{Sys}$**

$$R_{\text{QLOCK}} = \mathbf{Sys} = \{\text{init}\} \cup \{\text{want}(s,i) \mid s \in \mathbf{Sys}, i \in \text{Pid}\} \cup \{\text{try}(s,i) \mid s \in \mathbf{Sys}, i \in \text{Pid}\} \cup \{\text{exit}(s,i) \mid s \in \mathbf{Sys}, i \in \text{Pid}\}$$
Mutual exclusion property
as an invariant on $R_{QLOCK} = Sys$

```plaintext
mod INV1 {
  inc(QLOCK)
  -- declare a predicate to verify to be an invariant
  pred inv1 : Sys Pid Pid
  -- variables
  var S : Sys .
  vars I J : Pid .
  -- define inv1 to be the mutual exclusion property
  eq inv1(S,I,J)
      = (((pc(S,I) = cs) and (pc(S,J) = cs)) implies I = J) .
}
```

Formulation of proof goal for mutual exclusion property

$$INV1 \models \forall s \in Sys \forall i, j \in Pid. inv1(s, i, j)$$
Theorem of constants (TC)

\[ \text{INV1} \models \forall s \in \text{Sys} \forall i, j \in \text{Pid}. \text{inv1}(s, i, j) \]

||def

\[ \text{INV1} \models \text{inv1}(s: \text{Sys}, i: \text{Pid}, j: \text{Pid}) \]

||TC

\[ \text{INV1} \cup \{\text{op s : -> Sys}\} \cup \{\text{ops i j : -> Pid}\} \models \text{inv1}(s, i, j) \]

All the above three state:

for any model (or INV1-algebra) and
for any objects \( s: \text{Sys}, i: \text{Pid}, j: \text{Pid} \),
the proposition \( \text{inv1}(s, i, j) \) holds.
Constants and variables: Differences
(model/interpretation is assumed to be fixed
and $p$ is assumed to contain no variables in $S,E|= p$)

A **variable** of the same name which appears in the same equation in $S,E|= p$ denotes arbitrarily but the same object.

A **variable** of the same name which appears in several different equations in $S,E|= p$ denotes any object independently, and does not necessarily denote the same object.

A **constant** of the same name which appears in several different places in $S,E|= p$ denotes the same object, because a constant constitutes the signature $S$. 

Assertion and Proof Passage

**Assertion**

INV1  \( \cup \)

\{op s : \rightarrow Sys\}  \( \cup \)

\{ops i j : \rightarrow Pid\}

\|= in1(s,i,j)

**Logical Statement**

of stating that
Specification satisfies
property

**Proof Passage**

```plaintext
--  \([mx]*\)
open INV1
op s : \rightarrow Sys .
ops i j : \rightarrow Pid .
--  \|= red inv1(s,i,j) .
close
```

**Logical Statement and CafeOBJ Code**

If reduction part of the CafeOBJ code returns \texttt{true} then
the assertion holds.
If \([mx]^{*}\) returns \texttt{true} the verification is done, but ...

\[
\begin{aligned}
\text{-- } [mx]^{*} \\
\text{open INV1} \\
\quad \text{op } s : \to \text{ Sys . } \\
\quad \text{ops } i j : \to \text{ Pid .} \\
\text{-- } |= \text{ red inv1(s,i,j) .} \\
\text{close}
\end{aligned}
\]

This assertion states that mutual exclusion property holds for any reachable states.

If this proof passage returns \texttt{true} the game is over. But it does not return \texttt{true}.

A name of assertion/proof-passage(p.-p.) which ends with the character \(^{*}\) (like \([mx]^{*}\)) indicates that it does not return \texttt{true}. An assertion/p.-p. which return \texttt{true} is called to be \texttt{effective}. An effective p.p. has a name without \(^{*}\) at the end.
The goal of verification of an assertion $A$ via proof score is to get a set of assertions/p.-p.s $\{A_1, A_2, \ldots, A_n\}$ such that:

1. $(A_1$ and $A_2$ and $\ldots$ and $A_n)$ implies $A$, and
2. All the $A_1, A_2, \ldots, A_n$ are effective.

A set of assertions/p.-p.s $\{A_1, A_2, \ldots, A_n\}$ which satisfies (1) is called **proof score** (in a narrow sense) for $A$, and is also called **effective proof score** if it also satisfies (2).
For constructing proof scores, several kinds of assertion splitting rules of the form:

\((A_{i1} \text{ and } A_{i2} \text{ and } \ldots \text{ and } A_{in}) \implies A_i\)

are used. An assertion splitting rule is also written as:

\(\{A_{i1}, A_{i2}, \ldots, A_{in}\} \implies A_i\).

If \(n = 1\) the assertion splitting rule is also called assertion transformation rule and is written like:

\(A_{i1} \implies A\).
$R_{\text{QLOCK}}$ (set of reachable states) of $\text{OTS}_{\text{QLOCK}}$ (OTS defined by the module QLOCK)

**Signature determining $R_{\text{QLOCK}} = \text{Sys}$**

--- initial state
- op init : $\rightarrow \text{Sys}$ {constr}

--- actions
- bop want : $\text{Sys} \times \text{Pid} \rightarrow \text{Sys}$ {constr}
- bop try : $\text{Sys} \times \text{Pid} \rightarrow \text{Sys}$ {constr}
- bop exit : $\text{Sys} \times \text{Pid} \rightarrow \text{Sys}$ {constr}

**Recursive definition of $R_{\text{QLOCK}}$**

$R_{\text{QLOCK}} = \text{Sys} = \{\text{init}\} \cup$

$\{\text{want}(s,i) | s \in \text{Sys}, i \in \text{Pid}\} \cup$

$\{\text{try}(s,i) | s \in \text{Sys}, i \in \text{Pid}\} \cup$

$\{\text{exit}(s,i) | s \in \text{Sys}, i \in \text{Pid}\}$
Induction Scheme (I.S.)
induced by the structure of $R_{\text{QLOCK}} = \text{Sys}$

$$mx(s) \overset{\text{def}}{=} \forall i, j \in \text{Pid.inv1}(s, i, j)$$

\[
\begin{align*}
\{ & (\text{INV1} \models mx(\text{init})), \\
& (\text{INV1} \cup \{mx(s) = \text{true}\} \models \forall k \in \text{Pid.mx}(\text{want}(s, k))), \\
& (\text{INV1} \cup \{mx(s) = \text{true}\} \models \forall k \in \text{Pid.mx}(\text{try}(s, k))), \\
& (\text{INV1} \cup \{mx(s) = \text{true}\} \models \forall k \in \text{Pid.mx}(\text{exit}(s, k))) \} \\
\text{implies} \\
(\text{INV1} \models \forall s \in \text{Sys}.mx(s))
\end{align*}
\]

Induction Scheme (Assertion Splitting via I.S.)

\[
\{[1-\text{init}], [1-\text{want}]*, [1-\text{try}]*, [1-\text{exit}]*\}
\text{implies} [\text{mx}]*
\]
Assertion Splitting via Case Splitting

Because

INV1 |- c-want(s,k) or ~c-want(s,k)

Holds, the following assertion splitting is justified.

Assertion Splitting via Case Splitting

{[1-want,c-w]*, [1-want,~c-w]} implies [1-want]*

(CS)  { (E |= (p_1 or p_2)), (E U {p_1=true} |= p) ,
       (E U {p_2=true} |= p) }

implies  E |= p
**Meta Level Equation and Object Level Equation**

\[(E Q) \ E U \{ t_1 = t_2 \} \models p \ \text{iff} \ \ E U \{ ( t_1 = t_2 ) = \text{true} \} \models p\]

\((p = \text{true}) \ \text{iff} \ p\)

```latex
\begin{align*}
\rightarrow &\ [l\text{-want},c\text{-w-org}]^{*} \\
&\text{open INV1} \\
&\quad \text{op s} : \rightarrow \text{Sys} . \\
&\quad \text{ops i j k} : \rightarrow \text{Pid} . \\
&\quad \text{eq invl}(s,I:\text{Pid},J:\text{Pid}) = \text{true} . \\
&\quad \text{eq c-want}(s,k) = \text{true} . \\
&\quad \models \text{red invl}(\text{want}(s,k),i,j) . \\
&\text{close}
\end{align*}
```

\begin{align*}
\rightarrow &\ [l\text{-want},c\text{-w}]^{*} \\
&\text{open INV1} \\
&\quad \text{op s} : \rightarrow \text{Sys} . \\
&\quad \text{ops i j k} : \rightarrow \text{Pid} . \\
&\quad \text{eq invl}(s,I:\text{Pid},J:\text{Pid}) = \text{true} . \\
&\quad \text{eq c-want}(s,k) = \text{true} . \\
&\quad \models \text{pc}(s,k) = \text{rm} . \\
&\quad \models \text{red invl}(\text{want}(s,k),i,j) . \\
&\text{close}
\end{align*}
The proof passage \([1\text{-want, } c\text{-w, } \sim i=k, \sim j=k]*\) returns a Boolean term which is equivalent to \(\text{inv1}(s,i,j)\). This should return \text{true} because the Boolean term is an instance of \(\text{inv1}(s,\text{I:\text{Pid}}, \text{J:\text{Pid}})\) which is declared in the premise part (the part before \(|=\)) of this proof passage. This assertion-proof-passage is transformed to the effective one (the one which return \text{true}) by using \text{INST, TRANS, and HIDE} transforming rules.
-- [1-want,c-w,~i=k,~j=k]*
open INV1
  op s : -> Sys . ops i j k : -> Pid .
eq invl(s,I:Pid,J:Pid) = true .
  -- eq c-want(s,k) = true .
eq pc(s,k) = rm .
eq (i = k) = false .
eq (j = k) = false .
-- |=
  red invl(want(s,k),i,j) .
close

iff

-- [1-want,c-w,~i=k,~j=k,inst]*
open INV1
  op s : -> Sys . ops i j k : -> Pid .
eq invl(s,I:Pid,J:Pid) = true .
eq invl(s,i,j) = true .  **
  -- eq c-want(s,k) = true .
eq pc(s,k) = rm .
eq (i = k) = false .
eq (j = k) = false .
-- |=
  red invl(want(s,k),i,j) .
close
TRANS, HIDE

-- [I-want,c-w,~i=k,~j=k,inst,trans]*
open INV1
  op s : -> Sys . ops i j k : -> Pid .
  eq inv1(s,I:Pid,J:Pid) = true .
  -- eq c-want(s,k) = true .
  eq pc(s,k) = rm .
  eq (i = k) = false .
  eq (j = k) = false .
-- |
  red inv1(s,i,j) implies
  inv1(want(s,k),i,j) . **
close

including comments

iff

TRANS

excluding comments

implies

HIDE

Comment-out-ed equation is effective as “assertion”

-- [I-want,c-w,~i=k,~j=k,inst,trans,hide]
open INV1
  op s : -> Sys . ops i j k : -> Pid .
  -- eq inv1(s,I:Pid,J:Pid) = true . **
  -- eq c-want(s,k) = true .
  eq pc(s,k) = rm .
  eq (i = k) = false .
  eq (j = k) = false .
-- |
  red inv1(s,i,j) implies
  inv1(want(s,k),i,j) .
close
Some basic properties of $E |- p$ (1)

(term or equation which moves across $|$-

is assumed to include no variables)

Let $t_1'$ and $t_2'$ be the terms obtained from terms $t_1$ and $t_2$ by replacing variables in $t_1$ and $t_2$ with corresponding ground terms respectively then:

(INST) \[(E U \{t_1=t_2\})|= p \iff (E U \{t_1=t_2\} U \{t_1'=t_2'\})|= p\]

(TRANS) \[E |= (( t_1 = t_2 ) \implies p) \iff E U \{ t_1 = t_2 \} |= p\]
Some basic properties of $E |- p$ (2)
(term or equation which moves across |- is assumed to include no variables)

(HIDE) $(E \cup \{t1=t2\})|= p$ implies $(E \cup \{t1=t2\} \cup \{t3=t4\})|= p$

This justifies to comment out any equation (removing by making it comment) at any moment. (as a p.-p.)

(IMP) $E |= (t_1 = t_2)$ implies

$(E \cup \{t_1 = t_2\})|= p$ iff $E |- p$)
module ISTEP1

red inv1(s,i,j) implies inv1(want(s,k),i,j) .

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

red istep1(i,j).

Notice that using INST, TRANS, HIDE, and istep1,

ops k l : -> Pid . ... -- |=
    red inv1(s,k,l) implies istep1(i,j) .

can also be used instead of
    -- |=
    istep1(i,j)
for any k and l.
Simultaneous Induction Scheme

Lemma Discovery/Introduction:
Invariant inv2 is decided to be a lemma to be introduced.

Simultaneous Induction Scheme

and

Lemma Usage: Invariant predicate inv2 can be declared in the premise part of any assertion/proof-passage after the introduction.
Lemma declaration and its usage

-- [1-try2,c-t,i=k,~j=k,inv2]
open INV2
  ops i j k : -> Pid .
-- eq inv1(s,I:Pid,J:Pid) = true .
-- eq inv2(s,J:Pid) = true .
-- eq c-try(s,k) = true .
eq pc(s,k) = wt .
eq top(queue(s)) = k .
eq i = k .
eq (j = k) = false .
-- successor state
  eq s' = try(s,k) .
-- |=
  red inv2(s,j) implies istep1(i,j) .
close
No need to change already constructed assertions/proof-passages by lemma introduction

```lml
-- [1-want]*
open INV1
  op s : -> Sys .
  ops i j k : -> Pid .
  eq inv1(s,I:Pid,J:Pid) = true .
-- |=
  red inv1(want(s,k),i,j) .
close

-- [1-want2]*
open ISTEP2
  ops i j k : -> Pid .
  eq inv1(s,I:Pid,J:Pid) = true .
  eq inv2(s,I:Pid) = true .
  eq s' = want(s,k) .
-- |=
  red istep1(i) .
close
```
Final Proof Score for QLOCK

[1-init]
[1-want,c-w,i=k]
[1-want,c-w,~i=k,j=k]
[1-want,c-w,~i=k,~j=k,istep1]
[1-want,~c-w]
[1-try,c-t,i=k,j=k]
[1-2-try,c-t,i=k,~j=k,inv2]
[1-2-try,c-t,~i=k,j=k,inv2]
[1-2-try,c-t,~i=k,~j=k]
[1-try,~c-t]
[1-2-exit,c-e,i=k]
[1-2-exit,c-e,~i=k,j=k]
[1-2-exit,c-e,~i=k,~j=k]
[1-2-exit,~c-e]

[2-init]
[2-2-want,c-w,i=k]
[2-2-want,c-w,~i=k,queue(s)=empty]
[2-2-want,c-w,~i=k,queue(s)=j,q]
[2-2-want,~c-w]
[2-2-try,c-t,i=k]
[2-2-try,c-t,~i=k]
[2-2-try,~c-t]
[2-2-exit,c-e,,i=k]
[2-2-exit,c-e,~i=k,pc(s,i)=cs,inv1]
[2-2-exit,c-e,~i=k,~pc(s,i)=cs]
[2-2-exit,~c-e]