Lecture 4

The B Method

Structuring Mechanisms for B Specifications - `INCLUDES`

Lecture Outline

- References
- Abstract Machine Notation (AMN)
- Generalised Substitution Language (GSL)
- Incremental Model Development
- Multiple Generalised Substitution
- The `INCLUDES` Mechanism

References

- [1] Abrial, J.-R., *The B Book - Assigning Programs to Meanings*, Cambridge University Press, 1996. (chapter 7)
- [2] Schneider, S., *The B-Method - An Introduction*, Palgrave Macmillan, Cornerstones of Computing series, 2001. (chapter 10)
- [3] Clearsy System Engineering, *AtelierB home page* `http://www.atelierb.eu/en/`
- [4] Clearsy System Engineering, *B Method home page* `http://www.methode-b.com/en/`

Abstract Machine Notation (AMN)

- Consists of a number of clauses allowing the definition of *abstract machines*: `MACHINE`, `CONSTRAINTS`, `SETS`, `CONSTANTS`, ...

MACHINE
  $M(X, x)$
CONSTRAINTS
  C
SETS
  S;
  $T = \{a, b\}$
(ABSTRACT_)CONSTANTS
  c
PROPERTIES
  P
DEFINITIONS
  D
(CONCRETE_)VARIABLES
  v
INVARIANT
  I
ASSERTIONS
  J
INITIALIZATION
  U
OPERATIONS
  $u \leftarrow O(w) \hat{=}$
    PRE
      Q
    THEN
      V
    END;
  $\ldots$
END

Generalised Substitution Language (GSL)

- Consists of a number of *generalised substitutions* allowing the definition of abstract machine operations
- Generalised substitutions are an extension of the usual substitutions used in mathematics, the effect of which consists in the transformation of formulas on which they are applied
- A generalised substitution S is defined by the way it transforms an arbitrary predicate P, the meaning of the new predicate being axiomatized
  - Notation $[S]P$, read "S establishes P"

Elementary Substitutions

Substitution (syntax)		**Matching axiom** (semantics)
$v := E$	(simple substitution)	$[v := E]\, R \;\Leftrightarrow\; R$ with all free occurrences of v replaced by E
skip	(no effect substitution)	[skip] $R \;\Leftrightarrow\; R$
$P \mid S$	(preconditioned substitution)	$[P \mid S]\, R \;\Leftrightarrow\; (P \wedge [S]\, R)$
$P \Longrightarrow S$	(guarded substitution)	$[P \Longrightarrow S]\, R \;\Leftrightarrow\; (P \Rightarrow [S]\, R)$
$S \,[]\, T$	(bounded choice substitution)	$[S \,[]\, T]\, R \;\Leftrightarrow\; ([S]\, R \wedge [T]\, R)$
$@\, x \,.\, S$	(unbounded choice substitution)	$[@\, x \,.\, S]\, R \;\Leftrightarrow\; (\forall\, x \,.\, [S]\, R)$, if $x \backslash R$

Syntactic Extensions of Elementary Substitutions

Syntax	Definition
BEGIN S END	S
$x := E \mid\mid y := F$	$x, y := E, F$
PRE P THEN S END	$P \mid S$
IF P THEN S ELSE T END	$(P \Longrightarrow S) \, [] \, (\neg P \Longrightarrow T)$
IF P THEN S END	IF P THEN S ELSE skip END
CHOICE S OR $\ldots$ OR T END	$S \, [] \, \ldots \, [] \, T$
VAR x IN S END	$@x \,.\, S$
ANY x WHERE P THEN S END	$@x \,.(P \Longrightarrow S)$
$x :\in E$	ANY z WHERE $z \in E$ THEN $x := z$ END, if $z \backslash E$

Syntactic Extensions of Elementary Substitutions

Syntax	Definition
SELECT P THEN S	CHOICE $P \Longrightarrow S$
WHEN Q THEN $T \ldots$	OR $Q \Longrightarrow T \ldots$
WHEN R THEN U	OR $R \Longrightarrow U$
END	END
SELECT P THEN $S \ldots$	SELECT P THEN $S \ldots$
WHEN Q THEN T	WHEN Q THEN T
ELSE U	WHEN $\neg(P \vee \ldots \vee Q)$ THEN U
END	END

Incremental Model Development

- In order to control the complexity of the specification process, it is essential to have some sort of structuring mechanism allowing models to be developed in an incremental way
- B allows for the state information contained in a specification to be factored out in a number of separate machines, each responsible with operations working on its corresponding part of the state
- Advantages:
  - Conceptually distinct parts of a system are understood and described separately
  - Each machine has its consistency verified separately, its reuse involving also a reuse of the associated proof activity
  - A good structuring may reduce the proof effort by factoring proof obligations into appropriate machines
- Features enabling incremental specification in B
  - GSL: Multiple Generalised Substitution
  - AMN: `INCLUDES/EXTENDS/PROMOTES`, `USES/SEES` clauses

Multiple Generalised Substitution

- It is a generalisation of the $||$ operator, introduced previously as a mere "syntactic sugar" for multiple simple substitutions
- It is the basic ingredient allowing to build large specifications
- Construct: $S||T$, read "S with T", where S and T are generalised substitutions supposed to work on two abstract machines M and N, working with the respective distinct variables x and y.
- There is no rule for calculating $[S||T]P$ from $[S]P$ and $[T]P$
- When occuring in proof obligations, the generalised substitution must be reduced to a form in which the parallel operator has been removed
- There are reduction rules (equivalences) used to move a parallel operator inside choices and conditionals, until reaching a point where it is only applied on simple assignments, which can be rewritten to remove it completely

Multiple Generalised Substitution (cont.)

- Basic reduction rules

$x := E \,\|\|\, y := F$	= $x, y := E, F$
$S \,\|\|\, T$	= $T \,\|\|\, S$
$S \,\|\|\, skip$	= S
$S \,\|\|\, (P \mid T)$	= $P \mid (S \,\|\|\, T)$
$S \,\|\|\, (P \Longrightarrow T)$	= $P \Longrightarrow (S \,\|\|\, T)$
$S \,\|\|\, (T \,[]\, U)$	= $(S \,\|\|\, T) \,[]\, (S \,\|\|\, U)$
$S \,\|\|\, (@z.T)$	= $@z.(S \,\|\|\, T)$, if $z \backslash S$

- Similar rules apply when elementary substitutions are replaced by their syntactic extensions

The `INCLUDES` Mechanism

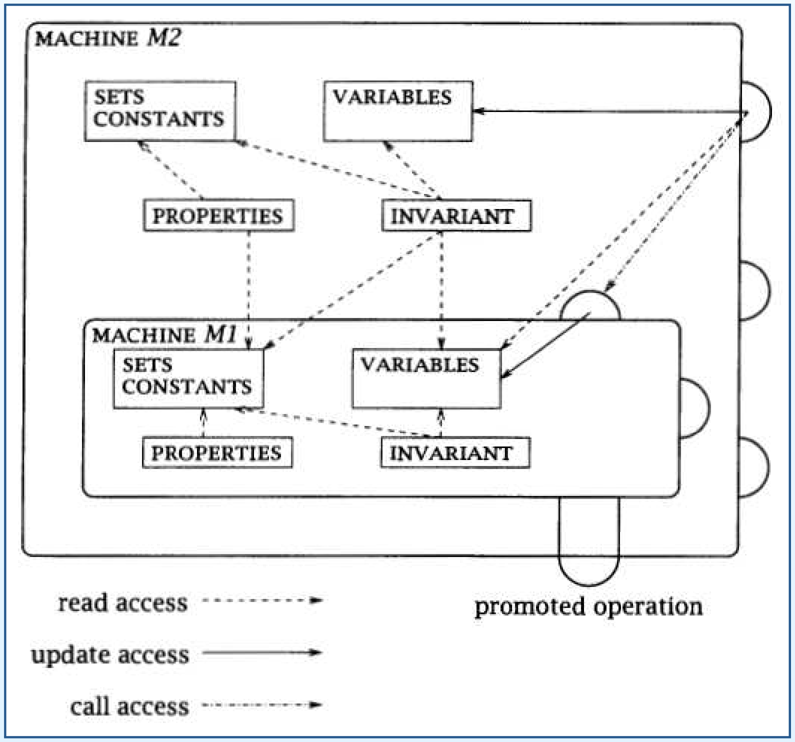
- Ensured by the AMN `INCLUDES` clause
  - A machine `M2` may include a previously built and proved machine `M1` by means of an `INCLUDES M1` statement written inside the description of `M2`
- Terminology
  - Information declared in `M2` = *native* information
  - Information from `M1` = *included* information
- If `M1` is parameterized, then its parameters should be instantiated upon inclusion in `M2`
  - The instantiation values should obey to the rules imposed by the `CONSTRAINTS` clause of `M1` (proof obligation in `M2`)
- The sets and constants of `M1` are visible to `M2`, as if they were declared in `M2`’s `SETS` and `CONSTANTS` clauses
  - The properties of `M2` can express constraints on included sets and constants (e.g. relations among its native and included sets and constants)

The `INCLUDES` Mechanism (cont.)

- The state of `M1` is part of the state of `M2`
  - The invariant of `M2` includes the invariant of `M1`
  - The invariant of `M2` can impose restrictions on the state of `M1` (e.g. properties relating its native and included variables)
- The operations of `M2` may access `M1`’s state in read mode in preconditions, guards and rhs of assignments
- The state of `M1` can only be altered by calls to `M1`’s operations from within `M2`’s operations
  - `M1` is the only responsible for preserving its invariant
  - Proof obligations are generated for `M2`, in order to ensure that the preconditions of called operations from `M1` are respected
- `M2` has complete control over `M1` (`M1` cannot be included into another machine)
  - Otherwise, the invariant of `M2` that refers to the state of `M1` may be broken

The `INCLUDES` Mechanism (cont.)

- The initialisation of `M2` first initializes the included machines, then executes its own statement
- `M1` should be defined independently from `M2`
- Graphical representation of the relationship between including and included machines [2]



Included Operations

- `M2`'s operation bodies are allowed to make "calls" to operations of `M1`
- The syntax of such an operation call is

  $$x_1, x_2, \ldots, x_n \leftarrow op(e_1, e_2, \ldots, e_m),$$

  where $e_1, e_2, \ldots, e_m$ are value expressions and $x_1, x_2, \ldots, x_n$ are distinct variables
- Such operation calls involve substituting in a substitution

Promotion and Extension

- `M1` is said to be completely under the control of `M2`: a state change in `M1` only happens through call of an operation of `M1` from within an operation of `M2`
- Operations of `M1` are all available to `M2`, but they are not automatically available to the environment of `M2` (they are not automatically part of the interface of `M2`)
- An operation of `M1` can be made available through the interface of `M2` by *promotion* (i.e. its name is listed in the `PROMOTES` clause of `M2`)
  - Promoted operations from `M1` must preserve the invariant of `M2` (proof obligation in `M2`)
- If all operations of `M1` are promoted by `M2`, then `M2` is said to be an *extension* of `M1` and should be declared as such by means of an extends clause (e.g. `MACHINE M2 EXTENDS M1` ...)
  - Extending a machine is equivalent to including it and promoting all its operations

Inclusion Example

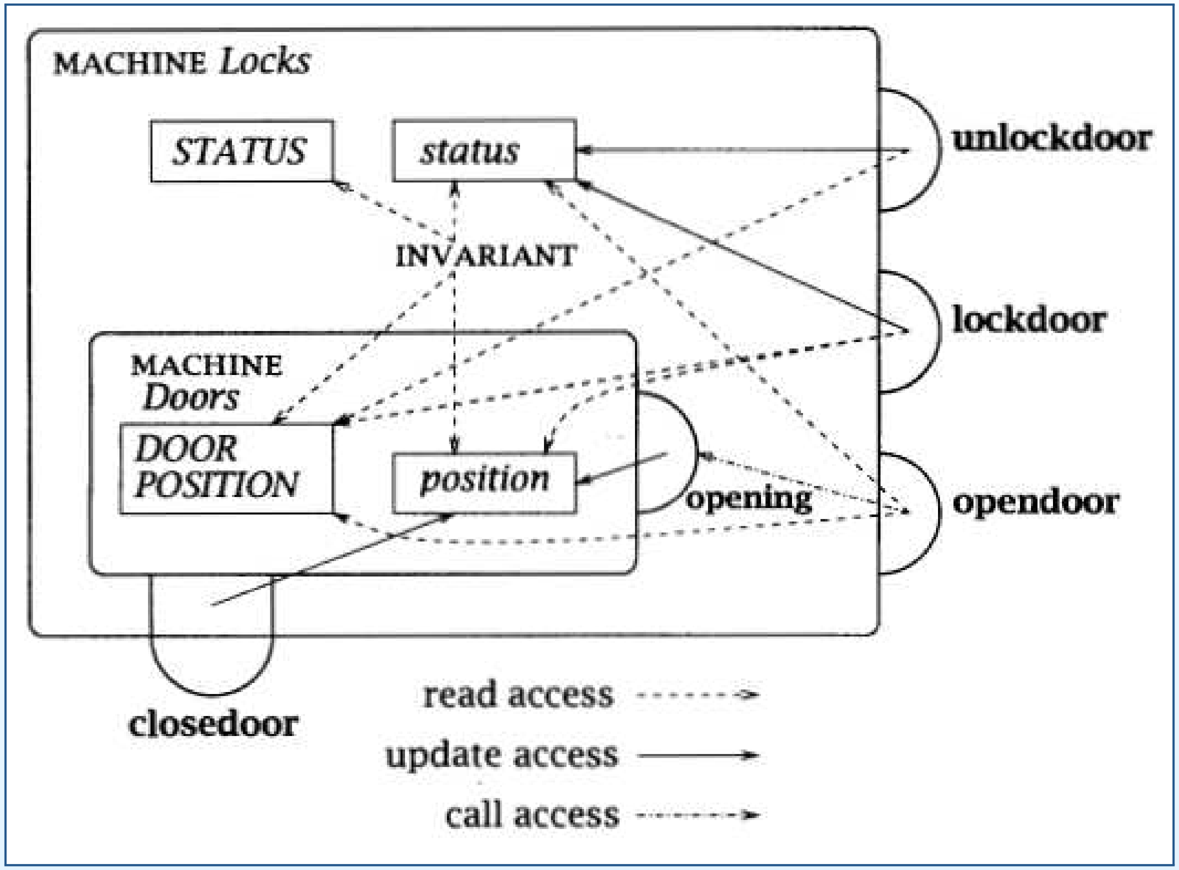
- Problem: Specify a system controlling the opening and closing of doors to safes in a bank vault. Required features:
  - open and close the doors
  - allow them to be locked and unlocked
- Machines `Doors` and `Locks`

```
MACHINE
    Doors
SETS
    DOOR;
    POSITION = {open, closed}
VARIABLES
    position
INVARIANT
    position ∈ DOOR → POSITION
INITIALISATION
    position := DOOR × {closed}
OPERATIONS
    opening(dd) =
    PRE
        dd ∈ DOOR
    THEN
        position(dd) := open
    END;

    closedoor(dd) =
    PRE
        dd ∈ DOOR
    THEN
        position(dd) := closed
    END
END
```

```
MACHINE
    Locks
INCLUDES
    Doors
PROMOTES
    closedoor
SETS
    STATUS = {locked, unlocked}
VARIABLES
    status
INVARIANT
    status ∈ DOOR → STATUS ∧
    position⁻¹[{open}] ⊆ status⁻¹[{unlocked}]
INITIALISATION
    status := DOOR × {locked}
OPERATIONS
    opendoor(dd) =
    PRE
        dd ∈ DOOR ∧ status(dd) = unlocked
    THEN
        opening(dd)
    END;

    lockdoor(dd) =
    PRE
        dd ∈ DOOR ∧ position(dd) = closed
    THEN
        status(dd) := locked
    END;

    unlockdoor(dd) =
    PRE
        dd ∈ DOOR
    THEN
        status(dd) := unlocked
    END
END
```

Inclusion Example (cont.)

- Graphical representation of the relationship between `Locks` and `Doors` [2]



Multiple Inclusion

- A machine can include a number of other machines, which, at their turn, can include other machines
- The includes relation is transitive, sets, constants and variables visibility being preserved through any number of inclusion levels
  - If `M3` includes `M2`, then `M3` has access to both the native and included information of `M2`
- Access to operations is not transitive though, unpromoted operations of a machine being only accessible in the machine that directly includes it
- When a machine includes several other machines, it may simultaneously modify some of their state by calling operations from these machines in parallel. Operations called in parallel should necessarily be from distinct machines.

Inclusion Example (cont.)

- Extension: Modify the previous example, so as to manage the locking and unlocking of doors by means of keys, with the following constraints
  - A key can only be inserted into a door whose lock it matches
  - A door should only be unlocked if there is a key present

- Machine Keys

```
MACHINE
    Keys
SETS
    KEY
VARIABLES
    keys
INVARIANT
    keys ⊆ KEY
INITIALISATION
    keys := ∅
```

```
OPERATIONS
    insertkey(kk) =
    PRE
        kk ∈ KEY
    THEN
        keys := keys ∪ {kk}
    END;

    removekey(kk) =
    PRE
        kk ∈ KEY
    THEN
        keys := keys - {kk}
    END
END
```

Inclusion Example (cont.)

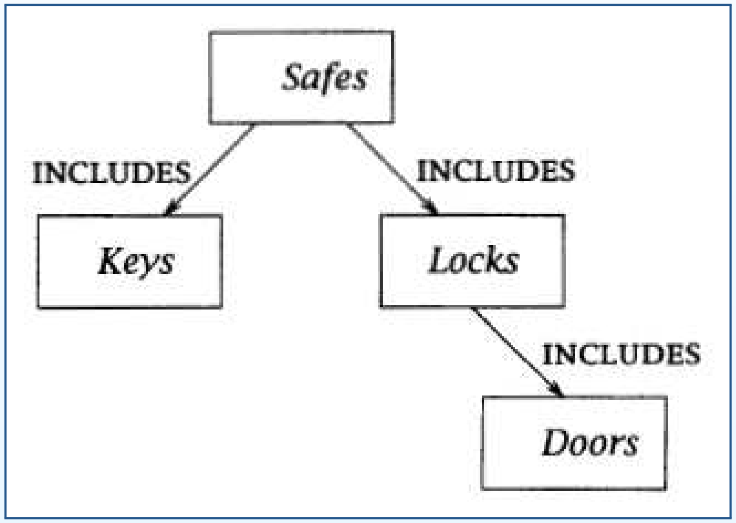
- Machine `Safes`

```
MACHINE
    Safes
INCLUDES
    Locks, Keys
PROMOTES
    opendoor, closedoor, lockdoor
CONSTANTS
    unlocks
PROPERTIES
    unlocks ∈ KEY ⤖ DOOR
INVARIANT
    status ⁻¹ [{unlocked}]  ⊆ unlocks[keys]
OPERATIONS
    insert(kk,dd) =
    PRE
        kk ∈ KEY ∧ dd ∈ DOOR ∧
        unlocks(kk) = dd
    THEN
        insertkey(kk)
    END;
```

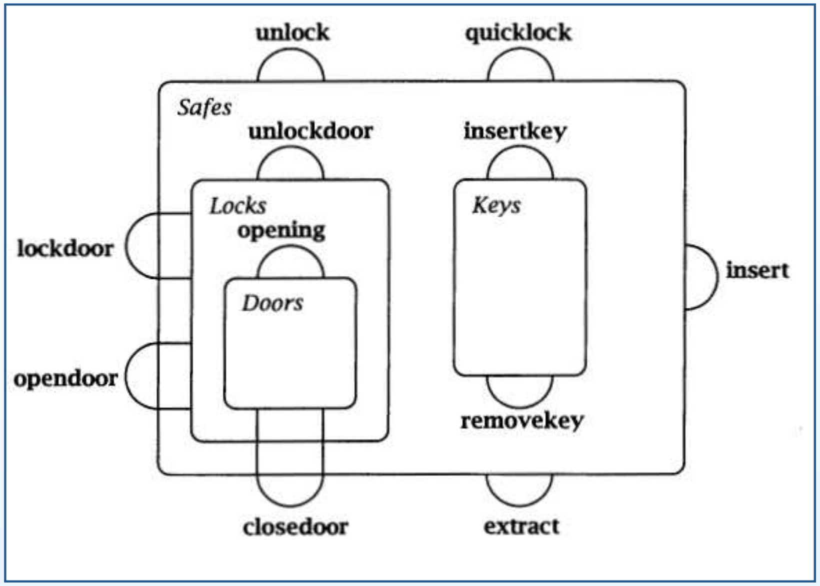
```
    extract(kk,dd) =
    PRE
        kk ∈ KEY ∧ dd ∈ DOOR ∧
        unlocks(kk) = dd ∧
        status(dd) = locked
    THEN
        removekey(kk)
    END;

    unlock(dd) =
    PRE
        dd ∈ DOOR ∧
        unlocks ⁻¹ (dd) ∈ keys
    THEN
        unlockdoor(dd)
    END;

    quicklock(dd) =
    PRE
        dd ∈ DOOR ∧
        position(dd) = closed
    THEN
        lockdoor(dd) ||
        removekey(unlocks ⁻¹ (dd))
    END
END
```

Inclusion Example (cont.)

- Machine relations [2]



- Final specification architecture [2]



General Inclusion Pattern

MACHINE $M_1(X_1, x_1)$ **CONSTRAINTS** C_1 **SETS** S_1; $T_1 = \{a_1, b_1\}$ **(ABSTRACT_)CONSTANTS** c_1 **PROPERTIES** P_1 **INCLUDES** $M(N, n)$ **(CONCRETE_)VARIABLES** v_1 **INVARIANT** I_1 **ASSERTIONS** J_1 **INITIALIZATION** U_1 **OPERATIONS** $u_1 \leftarrow O(w_1) \hat{=}$ **PRE** Q_1 **THEN** V_1 **END**; ... **END**	**MACHINE** $M(X, x)$ **CONSTRAINTS** C **SETS** S; $T = \{a, b\}$ **(ABSTRACT_)CONSTANTS** c **PROPERTIES** P **(CONCRETE_)VARIABLES** v **INVARIANT** I **ASSERTIONS** J_1 **INITIALIZATION** U **OPERATIONS** ... **END**

Inclusion Proof Obligations

- From top to bottom, the given POs ensure:
  1. The parameter values provided for the included machine fulfill its constraints
  2. Assertions of the including machine may be appropriately deduced
  3. Initialization of the including machine ensures its invariant
  4. Operations of the including machine preserve its invariant

$$\overbrace{A_1 \wedge B_1 \wedge C_1 \wedge P_1}^{Including} \Rightarrow \overbrace{[X, x := N, n](A \wedge C)}^{Included}$$

$$\overbrace{A_1 \wedge B_1 \wedge C_1 \wedge P_1 \wedge I_1}^{Including} \wedge \overbrace{B \wedge P \wedge [X, x := N, n](I \wedge J)}^{Included} \Rightarrow J_1$$

$$\overbrace{(A_1 \wedge B_1 \wedge C_1 \wedge P_1)}^{Including} \wedge \overbrace{(B \wedge P)}^{Included} \Rightarrow \overbrace{[[X, x := N, n]U]}^{Included} [U_1] I_1$$

$$\overbrace{(A_1 \wedge B_1 \wedge C_1 \wedge P_1 \wedge I_1 \wedge J_1 \wedge Q_1)}^{Including} \wedge$$
$$\overbrace{(B \wedge P \wedge [X, x := N, n] (I \wedge J))}^{Included} \Rightarrow [V_1] I_1$$

- Contextual abbreviations A, A_1, B, B_1 are defined as in Lecture 2.

Inclusion Example (cont.)

- Problem: Analyse for validity the proof obligation associated with the operation `quicklock` of `Safes`
- We should check the following proof obligation

$$
\begin{array}{l}
C \wedge B \wedge I \wedge dd \in DOOR \wedge position(dd) = closed \\
\quad \Rightarrow \quad [lockdoor(dd) \parallel removekey(unlocks^{-1}(dd))] I_{Safes}
\end{array}
$$

  where C is the conjunction of all native and included constraints clauses (none available), B is the conjunction of all native and included property clauses, and I is the conjunction of all native and included invariant clauses.

- We first expand and reduce the parallel operation calls

$$
\begin{array}{lll}
\multicolumn{3}{l}{lockdoor(dd) \parallel removekey(unlocks^{-1}(dd))} \\
= & \multicolumn{2}{l}{\textbf{PRE } dd \in DOOR \wedge position(dd) = closed} \\
 & \multicolumn{2}{l}{\textbf{THEN } status(dd) := locked \textbf{ END}} \\
 & \parallel & \textbf{PRE } unlocks^{-1}(dd) \in KEY \\
 & & \textbf{THEN } keys := keys - \{unlocks^{-1}(dd)\} \textbf{ END}
\end{array}
$$

Inclusion Example (cont.)

$$\begin{array}{ll} = & \textbf{PRE } dd \in DOOR \wedge position(dd) = closed \wedge unlocks^{-1}(dd) \in KEY \\ & \textbf{THEN } status(dd) := locked \parallel keys := keys - \{unlocks^{-1}(dd)\} \\ & \textbf{END} \\ = & \textbf{PRE } dd \in DOOR \wedge position(dd) = closed \wedge unlocks^{-1}(dd) \in KEY \\ & \textbf{THEN } status, keys := status \mathbin{\lhd\!\!-} \{dd \mapsto locked\}, keys - \{unlocks^{-1}(dd)\} \\ & \textbf{END} \end{array}$$

- The consequent of our proof obligation becomes

$$\begin{array}{l} dd \in DOOR \wedge position(dd) = closed \wedge unlocks^{-1}(dd) \in KEY \\ \wedge \; [status, keys := status \mathbin{\lhd\!\!-} \{dd \mapsto locked\}, keys - \{unlocks^{-1}(dd)\}] \\ \qquad\qquad (status^{-1}[\{unlocked\}] \subseteq unlocks[keys]) \end{array}$$

- The first three conjuncts are ensured by the antecedent of our proof obligation, while the last is ensured by the invariant itself

$$\begin{array}{ll} \multicolumn{2}{l}{status^{-1}[\{unlocked\}] \subseteq unlocks[keys]} \\ \Rightarrow & status^{-1}[\{unlocked\}] - \{dd\} \subseteq unlocks[keys] - \{dd\} \\ \Rightarrow & (status \mathbin{\lhd\!\!-} \{dd \mapsto locked\})^{-1}[\{unlocked\}] \\ & \qquad \subseteq unlocks[keys - \{unlocks^{-1}(dd)\}] \\ = & [status, keys := status \mathbin{\lhd\!\!-} \{dd \mapsto locked\}, keys - \{unlocks^{-1}(dd)\}] \\ & \qquad (status^{-1}[\{unlocked\}] \subseteq unlocks[keys]) \end{array}$$