Assuming a possible signature of M (e.g., Fig. 4.h) is "(>boolean, >example.DoorState, >T)", where the argument type to decide is prefixed with a tag ">", one decided value of the block Fig. 4.i can be "(true, state, new T(>N))". The variable "state" is put into a proper position that suits its type. The undecided (tagged) types are recursively decided until no tagged types exist. Therefore, the constructor of T, whose signature is "(>N)", continues to be decided. One decided value can be "(new N())". In practice, a type T can be a primitive, a class, an interface, or an array. For primitives (e.g., boolean), direct values are provided. For each interface or class type, the values are provided to allocate the type's concrete classes (similar to the "concrete" instruction based on isA relation). For each array type, e.g., T[][], the value provided is "new T[][] { >T[] }", which is recursively decided (i.e., "new T[][] { new T[] { >T } }" in the next expansion). The argument decision algorithm is as follows.

```
procedure ArgumentBlock::valueIterator(Context context)
    // assume that "example.Door" has a constructor whose parameter types is:
    //   "boolean, example.DoorState, T[], W" (T and W present interfaces or classes)
    // and a variable "state" whose type is "example.DoorState" is passed to this block

    // instantiate a value iterator that delegates hasNext() and next() request to this block
    valueIterator := prepareIteration(context)
    // get the types of this block's parameters, e.g., the type of the passed in parameter "state" is "example.DoorState"
    paramTypes := getParamTypes()

    // 1, search methods according to parameter types,
    //   i.e., the methods one of whose parameter types is the supper interface or superclass of "example.DoorState"
    // (The previous block provides the class that contains the methods to search)
    matchedMethods := search(paramTypes)

    // 2, decide the position of input params and tag other parameters of the matched methods
    // (The parameter "state" is put into the right position.
    //   If the method has other parameters, their types will be prefixed with ">")
    markedArguLines := assem(matchedMethods, paramTypes)
```

```
    // 3, decide one argument line by instantiating tagged types to prepare a next value
    decideOneArguLine(markedArguLines)

    return valueIterator
end procedure

procedure ArgumentBlock::decideOneArguLine(markedArguLines)
    // The initial element of the stack markedArguLines is “>boolean, state, >T[], >W”

    while not markedArguLines.isEmpty() then
        arguToDecide := markedArguLines.pop()
        decidedArgu := decideMarkedArguLine(arguToDecide, markedArguLines)
        if decidedArgu != null then
            return new BlockValue(decidedArgu)
        end if
    end while

    return null
end procedure

procedure ArgumentBlock::decideMarkedArguLine(arguToDecide, markedArguLines)
    // untag the first tagged type
    decidedArgu := decideTheFirstTaggedType(arguToDecide)

    // A possible value of the returned decidedArgu is “true, state, >T[], >W”.
```

```
    // The array type will be replaced with “new T[] { >T }” to be decided recursively.
    // Deciding a tagged type means to provide an object, e.g., by replacing “>W” with “new W()”.
    // If the constructor of W has parameters, they will be decided recursively.
    // Flexible strategies can be applied based on the isA relation to search the subclasses of W, which can be upcasted to W.
    // In addition to “new” an object, the object can be initialized based on closure, e.g., by replacing “>W” with:
    // new java.lang.Object() {
    //   public W func() {
    //     W obj = new W();
    //     obj.initialize();
    //     return obj;
    //   }
    // }.func()

    // push into the stack if decidedArgu still contains tagged types
    if containsTaggedTypes(decidedArgu) then
        markedArguLines.push(decidedArgu)
        return null
    end if

    return decidedArgu
end procedure
```