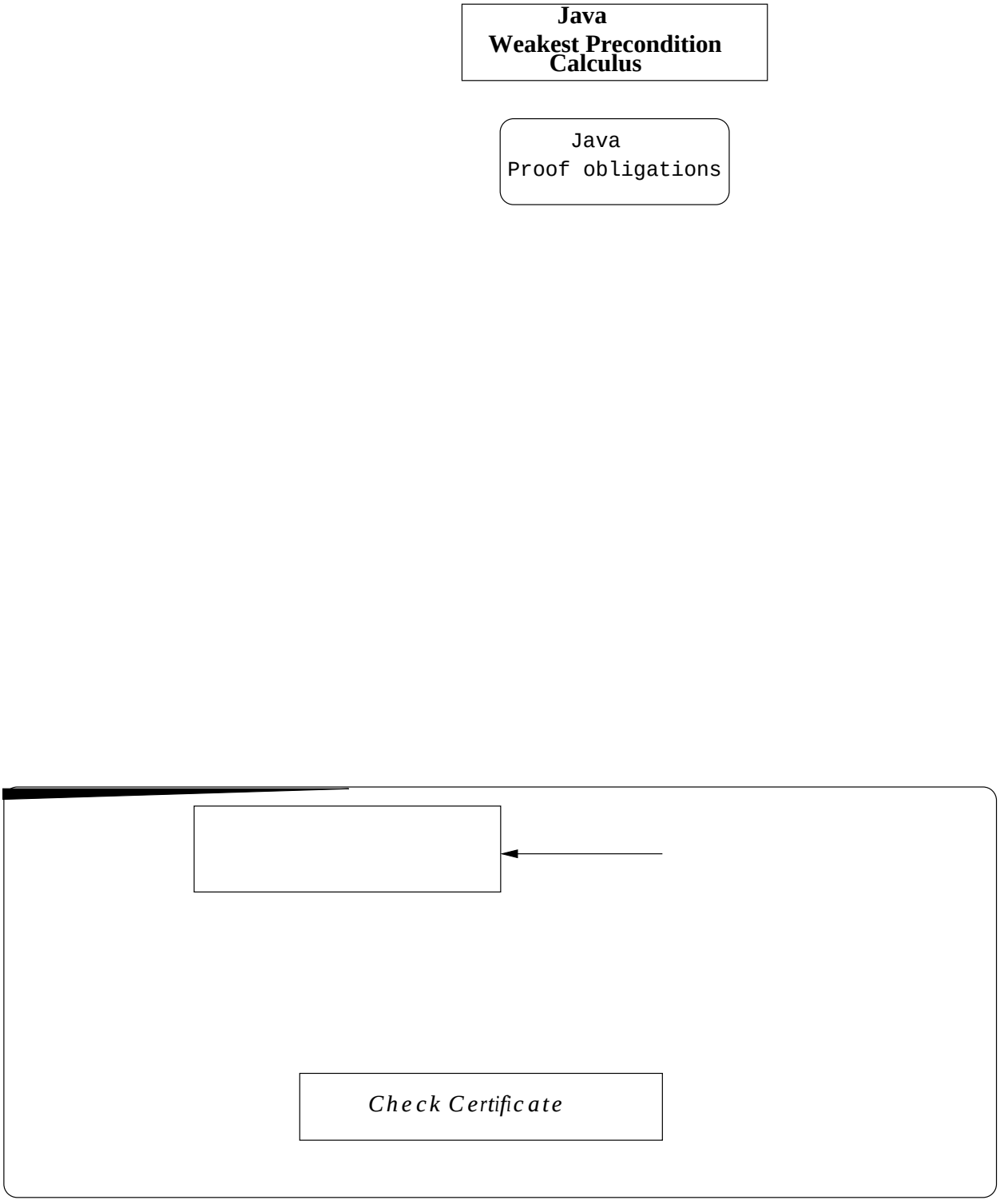
# SV-116
# Java Bytecode Speci cation and Veri cation

September 15,

**a bytecode speci cation language and a compiler from source program annotations into bytecode annotations. Thus, bytecode can bene t from the source speci cation and does not need to be accompanied by its**


Java
Weakest Precondition
Calculus
Java
Proof obligations
Check Certificate

approach presented here, where all specification clauses including loop invariants are compiled from the high level JML specification (see section 4.2).

The traditional PCC and the certifying compiler proposed by Necula (see [14, 15]) is an architecture for establishing trust in unknown code in which the code producer accompanies the code with a proof certificate. Differently from our approach, as the certifying compiler infers automatically a type specification such as loop invariants and generates automatically the proof certificate it is not applicable for complex security policies.

There are

```
public class ListArray {
  Object[] list;
  //@requires list != null;
  //@ensures \result == (\exists int i;0 <= i &&
            i < list.length && list[i] == o ) ;
  public boolean isElem(Object obj) {
    int i = 0;
    //@loop_modifies i;
    //@loop_invariant i <= &&
```

proving the soundness of the bytecode weakest precondition predicate transformer function becomes considerably complicated. This is not a restriction in the scenario in which we use BCSL speci cation | compiling JML to BCSL speci cation, as JML does not contain stack expressions.

## 4.2 Compiling JML into bytecode speci cation language

We now turn to

**LL**

**point among which loop invariants.**

**In Fig. 5, we show the $p$ rule for the `putfield` instruction. As the example shows the $p$ function takes three arguments: the instruction for which we calculate the precondition,p**

**statements, on bytecode**

We studied the relationship between the source code proof obligations generated by the standard feature of JACK and the bytecode proof obligations generated by our implementation over the corresponding bytecode produced by a non optimizing compiler over the examples given in [11]. The proof obligations were the same modulo program variables names and basic types.

We return now to our example

ordinal from previous sections and give in Fig. 7 one of the proof obligations on source and bytecode level respectively concerning the postcondition correctness. The verification conditions on bytecode and source level have

| | |
|---|---|
| lv[2]_ *t_ins_20* | i_ *t_ins_26* |
| *len*(#19( | |