# Formale Methoden im Softwareentwurf

**Summary**

Fabian Damken

March 6, 2022

TECHNISCHE UNIVERSITÄT DARMSTADT

# Contents

# 1 Introduction

## 1.1 Motivation

## 1.2 Traditional Reliability Measures

## 1.3 Classical Testing

## 1.4 Formal Methods

### 1.4.1 Specification

### 1.4.2 Difficulties

### 1.4.3 Abstraction and Expressiveness

### 1.4.4 Automatic Proof Search

## 1.5 Model Checking

## 1.6 Deductive Verification

## 1.7 Tool Support

# 2 PROMELA

*PROMELA* is an acronmy for *Process meta-language* and is a language for modeling concurrent systems:

- Multi-threaded
- Synchronization and message passing
- Only a few control structures and pure expressions with no side-effects
- Data structures with finite and fixed bound

But PROMELA is not a programming language and thus not intended to be used for programming real systems.

- No reference types,
- no methods or procedures (just macros),
- no libraries,
- no GUI nor standard input,
- no floating point types and
- no data encapsulation.
- Assumes fair scheduling policy during verification and is
- non-deterministic (but executable, e.g. with a random scheduler).

Listing 2.1: First PROMELA Program

```
1 active proctype P() {
2     printf("Hello, World!\n")
3 }
```

- The program 2.1 can be simulated (executed, interpreted) using SPIN (this will be covered in more detail in chapter 4):

```
> spin  hello-world.pml
      Hello, World!
1 process created
```

- The keyword `proctype` declares a *process* named P.
- Printing using `printf` uses a simplified C-like formatting syntax.
- PROMELA is general is a bit C-like (also adapts the comment style of C).

## 2.1 Data Types

### 2.1.1 Arithmetic

- The data types `byte`, `short`, `int` and `unsigned` have the operations `+`, `-`, `*`, `/` and `%` where the semantics are mirrored from the underlying C data types.
  - All expressions are calculates as `int` and are then converted into the container type.
  - Variables that have not been initialized explicitly default to `0`.
- No floats and no strings.
- The compiler moves all variable declarations to the start of the process, so avoid putting declarations anywhere else than the start!

### 2.1.2 Booleans

- The data type `bool` has the literals `true` and `false` which are only syntactic sugar for `0` (`false`) and `1` (`true`), thus is `bool` for `bit`.

### 2.1.3 Enumerations

Listing 2.2: PROMELA Enumerations

```
1 mtype = { red, yellow, green };
2 mtype light = green;
3 printf("The light is %e\n", light)
```

- The literals are names for a non-zero `byte`, thus at most 255 elements can be defined.
- The abbreviation `mtype` stands for *message type* (as the first use was for message names).
- It is only possible to define at most one `mtype` per program.

### 2.1.4 Arrays

Listing 2.3: PROMELA Arrays

```
1 /* Defines N as beeing 5, this is evaluated by the compiler. */
2 #define N 5
3
4 active proctype P() {
5     byte a[N];
6     byte i;
```

```
    byte sum = 0;
    a[0] = 0;  a[1] = 10;  a[2] = 20;  a[3] = 30;  a[4] = 40;
    for (i in a) {
        sum = sum + a[i]
    }
}
```

- Array indices start with zero, like in C.
- All entries are initialized with zero.
- Arrays are value types and `a` and `b` are always different (`a != b`).
- The bounds of an array (especially the length) are constant and cannot be changed (there are no unbound data types in PROMELA!).
- Only one-dimensional arrays are possible (except for ugly workarounds).
- For-loops like in program 2.3 run over the indices of the array, not the elements!

### 2.1.5 Record

Listing 2.4: PROMELA Records

```
typedef DATE {
    int day, month, year;
}

active proctype P() {
    DATE D;
    D.day = 11;  D.month = 11;  D.year = 1997
}
```

- Similar to structs in C.
- Record types may include references to previously defined record type, but no self-references are allowed!
- Can be used to realize multi-dimensional arrays (with arrays of record types including arrays)...

## 2.2 Control Statements

**Sequencing** Using a semicolon (`;`) as a separator. It is not required after the last statement.

**Guarded Commands** Selection: Non-deterministic choice of an alternative; Repetition: Loop until break, or forever.

**For-Loop** Translated to a do-loop, loops over keys of an array.

**Goto** Jumps to a label.

### 2.2.1 Selection

Listing 2.5: PROMELA Selection

```
active proctype P() {
    byte a = 5, b = 5;
    byte max, branch;
    if
        :: a >= b -> max = a; branch = 1
        :: a <= b -> max = b; branch = 2
    fi
    printf("Max: %d; Branch: %d\n", max, branch)
}
```

- The *guards* (the conditions before the `->`) may overlap. That is, multiple guard may be true at one time (like in program 2.5).
- Any alternative with a true guard is chosen randomly (non-deterministic).
- When no guard is true, the process blocks until one becomes true (if this never happens, that's a problem).
- During command line execution, the trace of the random simulation can be printed:

```
> spin -v selection.pml
spin: code/promela/selection.pml:0, warning, proctype P, 'byte  max' variable is never used (other
    than in print stmnts)
spin: code/promela/selection.pml:0, warning, proctype P, 'byte  branch' variable is never used
    (other than in print stmnts)
  1:    proc  0 (P:1) code/promela/selection.pml:5 (state 1)    [((a>=b))]
  2:    proc  0 (P:1) code/promela/selection.pml:5 (state 2)    [max = a]
  3:    proc  0 (P:1) code/promela/selection.pml:5 (state 3)    [branch = 1]
  4:    proc  0 (P:1) code/promela/selection.pml:8 (state 8)    [.(goto)]
      Max: 5; Branch: 1
  5:    proc  0 (P:1) code/promela/selection.pml:8 (state 9)    [printf('Max: %d; Branch:
      %d\\n',max,branch)]
1 process created
```

- The Symbol `->` is overloaded in PROMELA:
  - Also used in conditional expressions `boolean-guard -> then : else`.
  - The brackets are mandatory in conditional expressions.
- The first statement after `::` is evaluated as the guard. It is possible to use a semicolon instead of an arrow, but this is confusing and shall not be done.
- If an alternative should be selectable every time, it can be the literal `true`.
- To mark an alternative as selectable iff no other alternative is true, it should be marked with `else`.

### 2.2.2 Repetition

Listing 2.6: PROMELA Repetition

```
active proctype P() {
    int a = 15, b = 20;
    do
```

```
        :: a > b  -> a = a - b
        :: b > a  -> b = b - a
        :: a == b -> break
    od
}
```

- Like in the selection statement, any alternative with a true guard is chosen randomly.
- The only way to exit a loop is by `break` or `goto`.
- The command `skip` continues the loop in the next iteration.
- When no guard is true, the loop blocks until one becomes true.
- During command line execution, the trace of the random simulation can be printed as well as the values of the local variables:

```
> spin -p -l repetition.pml
  0:    proc  - (:root:) creates proc  0 (P)
  1:    proc  0 (P:1) code/promela/repetition.pml:5 (state 3)   [((b>a))]
  2:    proc  0 (P:1) code/promela/repetition.pml:5 (state 4)   [b = (b-a)]
        P(0):b = 5
  3:    proc  0 (P:1) code/promela/repetition.pml:8 (state 8)   [.(goto)]
  4:    proc  0 (P:1) code/promela/repetition.pml:4 (state 1)   [((a>b))]
  5:    proc  0 (P:1) code/promela/repetition.pml:4 (state 2)   [a = (a-b)]
        P(0):a = 10
  6:    proc  0 (P:1) code/promela/repetition.pml:8 (state 8)   [.(goto)]
  7:    proc  0 (P:1) code/promela/repetition.pml:4 (state 1)   [((a>b))]
  8:    proc  0 (P:1) code/promela/repetition.pml:4 (state 2)   [a = (a-b)]
        P(0):a = 5
  9:    proc  0 (P:1) code/promela/repetition.pml:8 (state 8)   [.(goto)]
 10:    proc  0 (P:1) code/promela/repetition.pml:6 (state 5)   [((a==b))]
 11:    proc  0 (P:1) code/promela/repetition.pml:6 (state 6)   [goto :b0]
 11:    proc  0 (P:1)       terminates
1 process created
```

## For-Loops

Listing 2.7: PROMELA For-Loops

```
#define N 10

active proctype P() {
    int i;
    int sum = 0;
    for (i : 1 .. N) {
        sum = sum + i
    }
}
```

- Starting from SPIN version 6, it is possible to have C-style for loops.
- They are translated into do-loops by the compiler and can iterate over Array indices (like in program 2.3) or over ranges (like in program 2.7).

### 2.2.3 Jumps

Listing 2.8: PROMELA Jumps

```
#define N 10

active proctype P() {
    int sum = 0;
    byte i = 1;
    do
        :: i > N -> goto exitloop
        :: else  -> sum = sum + i; i++
    od
    exitloop:
    printf("End of loop.\n")
}
```

- It is possible to define labels and jump to them using `goto`.
- These labels must be unique for a process and it is not possible to place labels in front of guards.
- It is easy to write messy code with `goto`! Take this into account when using!

## 2.3 Inlining Code

Listing 2.9: PROMELA Inlines

```
typedef DATE {
    int day, month, year;
}

inline setDate(D, DD, MM, YY) {
    D.day = D;  D.month = MM;  D.year = YY;
}

active proctype P() {
    DATE d;
    setDate(d, 11, 11, 1997);
}
```

- PROMELA has no methods or procedure calls, but it is possible to create macro-like abbreviations for code that appears multiply.
- The code is then inlined (i.e. copied) into the “calling” place.
- Thus, no new scope is created and all variables may be modified $\implies$ avoid declaring variables in and inline block, they are visible for the calling process.

## 2.4 Nondeterminism

Deterministic PROMELA programs are trivial: Assuming a program with only one process and no overlapping guards, all variables are (implicitly or explicitly) initialized as no user input is possible and each state is either blocking or has exactly one successor state. Non-trivial PROMELA programs are non-deterministic with different possible sources of non-determinism:

1. Arbitrary (non-deterministic) choice of alternatives with overlapping guards and
2. Scheduling of concurrent processes.

### 2.4.1 Value Generation

One of the most important sources of non-determinism is the non-deterministic generation of values. This section will cover some possibilities for such generations.

Listing 2.10: PROMELA Non-Deterministic Value Generation by Guards

```
active proctype P() {
    byte range;
    if
        :: range = 1
        :: range = 2
        :: range = 3
        :: range = 4
    fi
    printf("Chose %d\n", range)
}
```

- The code in listing 2.10 is equivalent to multiple Guards `:: true -> range = 1` etc. as an assignment is always true (it always succeeds).
- This yields a non-deterministic choice for `range` in the domain $\{1, 2, 3, 4\}$.

Listing 2.11: PROMELA Non-Deterministic Value Generation by Loop

```
#define LOW  0
#define HIGH 9

active proctype P() {
    byte range = LOW;
    do
        :: range < HIGH -> range++
        :: break
    od
    printf("Chose %d\n", range)
}
```

- The code in listing 2.11 generates a value for `range` in the domain { `LOW`, ··· , `HIGH` } with the probability $2^{-(n+1)}$ for generating the number $n$.
- This the probability for larger numbers reduced $\Longrightarrow$ does not yields representative test cases for random simulation.
- But as in verification, all computations are generated, this solution works well enough for verification.

Listing 2.12: PROMELA Non-Deterministic Value Generation by Select

```
#define LOW  0
#define HIGH 9

active proctype P() {
    int i;
    select(i: LOW .. HIGH);
    printf("Chose %d\n", i)
}
```

- The `select` macro is only available in SPIN version 6 and above.
- Syntactic sugar for generating values in a given range.

## 2.5 Concurrent Systems

- A concurrent system is about “doing things at the same time trying not to get into each others way”.
- Doing things at the same time can mean a lots of things, this course focuses on sharing computational resources, especially memory.
- The basic concepts for controlling the accesses are:
  - Blocking, locks
  - Semaphores
  - Busy waiting

  And these need to be carefully designed, otherwise a deadlock might happen!

### 2.5.1 Modeling in PROMELA

Listing 2.13: PROMELA with multiple Processes

```
active [2] proctype P() {
    printf("Process P%d, statement 1\n", _pid);
    printf("Process P%d, statement 2\n", _pid)
}

active [2] proctype Q() {
    printf("Process Q%d, statement 1\n", _pid);
    printf("Process Q%d, statement 2\n", _pid)
}
```

- The primary concept for modeling concurrent systems in PROMELA is the use of processes.
- In PROMELA, more than one process can be created (max. 255) that all need unique identifiers.
- Listing 2.13 shows a program with two processes (introduced by `proctype`).
- As only one process can be executed at one time on the processor, the processes are executed concurrently.
- The scheduler selects the next process randomly.
- This yields many different *interleavings* and thus non-determinism.
- The `[2]` between `active` and `proctype` declares how many processes of the type should be created. If no number is given, SPIN defaults to one process.
- The current process identifier is stored in the reserved variable `_pid`.
- One sample execution of the program 2.13 (the indicates the process that has generated the string, “one tab per process ID”; can be suppressed with the switch -T):

```
> spin  multiple-processes.pml
          Process P1, statement 1
      Process P0, statement 1
                  Process Q3, statement 1
          Process P1, statement 2
              Process Q2, statement 1
                  Process Q3, statement 2
              Process Q2, statement 2
      Process P0, statement 2
4 processes created
```

### 2.5.2 Starting Processing Explicitly

Listing 2.14: PROMELA Init Process

```
proctype P() {
    printf("In process P%d\n", _pid)
}

proctype Q() {
    printf("In process Q%d\n", _pid)
}

init {
    run P();
    run P();
    run Q()
}
```

- There is exactly one *initial process* that spawns the other ones.
- This process is often declared implicitly using the keyword `active` in the process definition (this process is ran once the program is started).
- The initial process can also be explicitly declared with the keyword `init`, as shown in listing 2.14. This programs spawns three processes, one instance of `P` and two of `Q`.
- Another process is started with the keyword `run`. This invocation does not wait until the process is ready, thus the code in process `Q` may be executed even before the first execution of an instance of process `P`.
- It is also possible to wrap the invocations of `run` into an `atomic { ... }` block, causing the process to execute only after all processes are created. The keyword `atomic` is discussed in detail in section 2.5.6.

#### Parameters

Processes may also have arguments that have to be passed when instantiated by `run`, see listing 2.15.

Listing 2.15: PROMELA Processes with Parameters

```
1 proctype P(byte i; bool b) { ... }
2
3 init {
4     run P(7, true);
5     run P(8, false)
6 }
```

### 2.5.3 Joining Processes

Listing 2.16: Joining PROMELA Processes

```
1 proctype P() { ... }
2
3 init {
4     atomic {
5         run P();
6         run P()
7     }
8     (_nr_pr == 1) -> printf("ready")
9 }
```

Processes may be joined by using a trick: Wait for the variable `_nr_pr` to equal `1`, indicating that only one process still runs (the variable holds the number of processes currently running). This utilization is shown in listing 2.16.

### 2.5.4 Local and Global Data

- Variables declared outside of any process are visible (and writable) for/by all processes.
- Variables declared inside a process are local to that process and only visible for that process.

- Pragmatics of modeling with global data:
  - *Shared memory* of concurrent systems are often modeled by global variables of numeric (or array) type.
  - *Shared resources* like the state of a printer, traffic light, etc. are often modeled by global variables of type boolean or enumeration type.
  - *Communication* the media of distributed systems is often modeled by global variables of type channel (`chan`), this is discussed in more detail in section 2.6.

  Global variables must never be used to model process-local data!

#### Interference of Global Data

Listing 2.17: PROMELA Global Data Interference

```
byte n = 0;

active proctype P() {
    n = 1;
    printf("Process P; n: %d\n", n)
}

active proctype Q() {
    n = 2;
    printf("Process Q; n: %d\n", n)
}
```

Given the program in listing 2.17, how many possible outputs exist? This can be drawn in "transition graph" shown in figure 2.1.

#### Synchronizing Global Data

PROMELA provides no primitives like semaphores, locks or monitors for synchronizing on global data. Instead, PROMELA controls the statement *executability* which is covered in more detail in section 2.5.7.

### 2.5.5 Computations

Listing 2.18: PROMELA Computations Example

```
active [2] proctype P() {
    byte n = 0;
    n = 1;
    n = 2
}
```

The possible computations of a PROMELA program can be shown in a transition graph. For example, figure 2.2 shows the graph for possible computations of the program 2.18 where only one computation is possible (as no global data is used). The program pointer (line number) for each process is shown in the left/upper compartment and the values of the variables in the lower compartment.

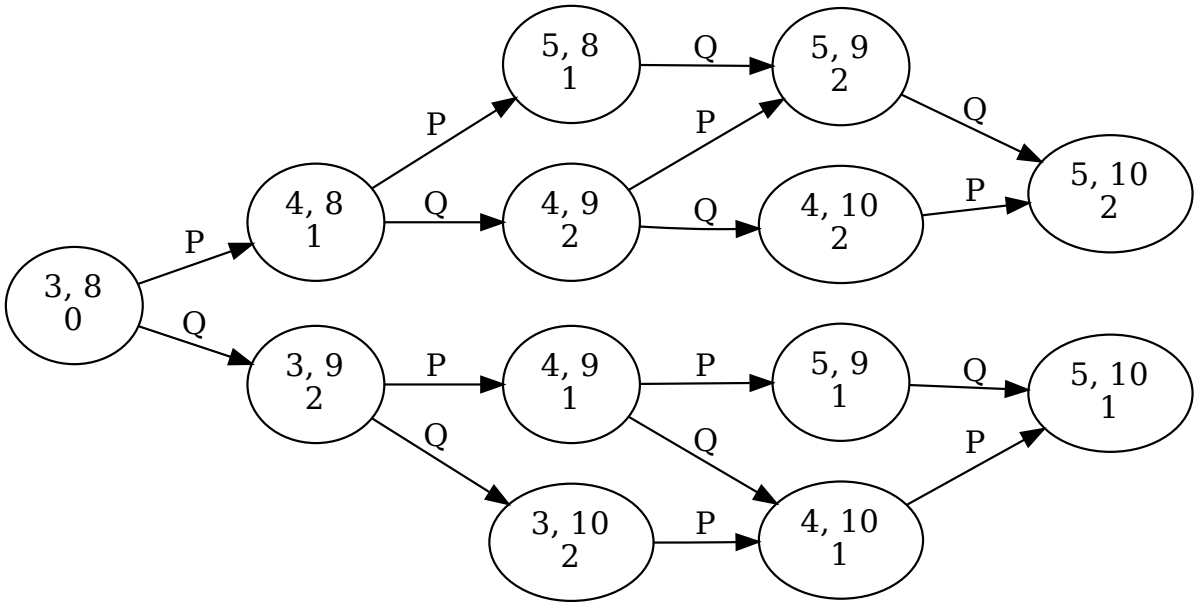


Figure 2.1: PROMELA Interference Graph

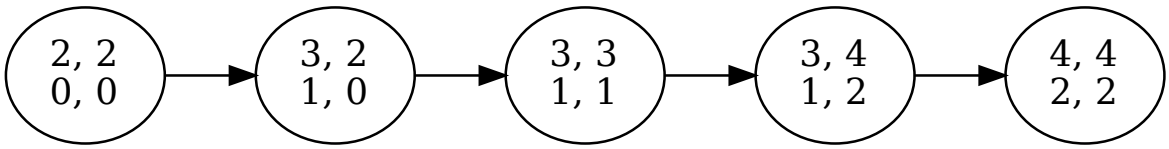


Figure 2.2: PROMELA Computations Example

## Interleaving

**Definition (Interleaving of independent Computations)** Assume $n$ independent processes $P_1, \cdots, P_n$ where process $P_i$ has the computation $c^i = (s_0^i, s_1^i, s_2^i, \cdots)$.

The computation $(s_0, s_1, s_2, \cdots)$ is an *interleaving* of $c^1, \cdots, c^n$ iff for all $s_j = s_{j'}^i$ and $s_k = s_{k'}^i$ with $j < k$ it is the case that $j' < k'$.

These interleavings can be represented as a directed graph. See figure 2.3 for the interleaving graph of the program 2.18.

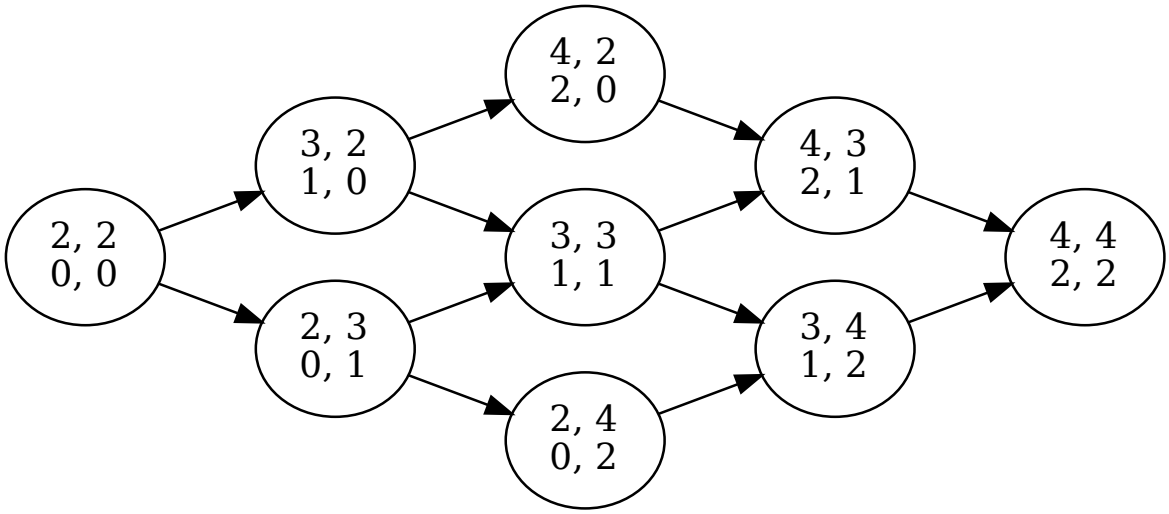


Figure 2.3: PROMELA Interleavings

### 2.5.6 Atomicity

**Definition (Atomicity)** An expression or statement that is executed entirely without the possibility of interleaving is called *atomic*.

- In PROMELA, assignments, jumps, skip and expressions are atomic.
- In particular, conditional expressions `(p -> q : r)` are atomic (brackets required).
- Guarded commands are not atomic! This the code in listing 2.19 is not save as a division by zero may occur in line 6 if process `Q` is executed after the evaluation of the guard `a != 0`, but before the execution of `c = b / a`. This can be enforced by using interactive simulation:

```
> spin -p -g -i divbyzero.pml
  0: proc  - (:root:) creates proc  0 (P)
  0: proc  - (:root:) creates proc  1 (Q)
Select a statement
    choice 1: proc  1 (Q:1) \lstbasepath/code/promela/divbyzero.pml:12 (state 1) [a = 0]
    choice 2: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:4 (state 1) [a = 1]
Select [1-2]: 2
  1: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:4 (state 1)  [a = 1]
       a = 1
Select a statement
    choice 1: proc  1 (Q:1) \lstbasepath/code/promela/divbyzero.pml:12 (state 1) [a = 0]
    choice 2: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:4 (state 2) [b = 1]
Select [1-2]: 2
  2: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:4 (state 2)  [b = 1]
       b = 1
Select a statement
    choice 1: proc  1 (Q:1) \lstbasepath/code/promela/divbyzero.pml:12 (state 1) [a = 0]
    choice 2: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:4 (state 3) [c = 1]
Select [1-2]: 2
```

```
  3: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:4 (state 3)  [c = 1]
        c = 1
Select a statement
    choice 1: proc  1 (Q:1) \lstbasepath/code/promela/divbyzero.pml:12 (state 1) [a = 0]
    choice 2: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:5 (state 8) [((a!=0))]
    choice 3: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:5 (state 8) unexecutable, [else]
Select [1-3]: 2
  4: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:6 (state 4)  [((a!=0))]
Select a statement
    choice 1: proc  1 (Q:1) \lstbasepath/code/promela/divbyzero.pml:12 (state 1) [a = 0]
    choice 2: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:6 (state 5) [c = (b/a)]
Select [1-2]: 1
  5: proc  1 (Q:1) \lstbasepath/code/promela/divbyzero.pml:12 (state 1) [a = 0]
        a = 0
Select a statement
    choice 1: proc  1 (Q:1) \lstbasepath/code/promela/divbyzero.pml:13 (state 2) <valid end state>
        [-end-]
    choice 2: proc  0 (P:1) \lstbasepath/code/promela/divbyzero.pml:6 (state 5) [c = (b/a)]
Select [1-2]: 2
[1]    2513 floating point exception  spin -p -g -i divbyzero.pml
```

Listing 2.19: Atomicity of Guards in PROMELA

```
int a, b, c;

active proctype P() {
    a = 1;  b = 1;  c = 1;
    if
        :: a != 0 -> c = b / a
        :: else   -> c = b
    fi
}

active proctype Q() {
    a = 0
}
```

## Deterministic Sequences

- A sequence marked with `d_step` is *strongly atomic* and cannot be interrupted.
- It is called a *deterministic sequence* and inside it, every non-determinism is avoided. That is, in the case of overlapping guards, the first alternative is chosen. Nevertheless, non-determinism should be avoided in `d_step`!
- If any statement in a `d_step` other than the first blocks, an error occurs.
- If the first statement (the “guard”) blocks, `d_step` is not entered as a whole.
- Listing **??** shows the general syntax.

Listing 2.20: Deterministic Sequence in PROMELA

```
d_step {
    stmt1;  /* Guard */
    stmt2;
    stmt3
}
```

| Statement Type | Executable |
|---|---|
| Assignments | always |
| Assertions | always |
| Print Statements | always |
| Expression Statements | iff value not `0`/`false` |
| `atomic`, `d_step` | iff the guard is executable |
| `if`, `do` | iff any alternative is executable |
| alternative of `if`, `do` | iff the guard is executable |
| `for` | always (body can block) |

Table 2.1: Basic Executability in PROMELA

#### Atomic Sequences

- A sequence marked with `atomic` is *weakly atomic* and can be interrupted if and only if a statement blocks.
- In an atomic block, non-determinism is possible (so overlapping guards have the well-known behavior of selecting a random alternative).
- Any statement can block inside an `atomic` block and the execution may be interleaved iff any statement blocks.
- The block is only entered iff the first statement (the "guard") does not block.
- Listing 2.21 shows the general syntax.

Listing 2.21: Atomic Sequence in PROMELA

```
atomic {
    stmt1;  /* Guard */
    stmt2;
    stmt3
}
```

### 2.5.7 Executability

- PROMELA does not have synchronization primitives like semaphores, locks or monitors.
- Instead, PROMELA controls a statements *executability* (the absence of blocking).
- Non-executable statements in atomic statements permit preemption.
- Most known synchronization primitives like test-and-set, compare-and-swap, semaphores, ... can be modeled using executability and atomicity.
- The executability of basic statements is shown in table 2.1.

### Blocking

**Definition (Blocking)** A *statement* is *blocking* iff it is not executable. A *process* is *blocking* iff its location counter points to a blocking statement.

- For the next step of the execution, the scheduler chooses one of the non-blocking statements randomly (non-deterministically).
- Executability and blocking are the basic concepts in PROMELA for modeling solutions to synchronization problems.

## 2.5.8 Critical Section Problem

The *critical section problem* is an archetypical problem of concurrent systems.

**Definition (Critical Section)** The *critical section* (CS) of a process is the block of code where shared state (e.g. global variables) are accessed and possible manipulated.

**Definition (CS Problem)** Given a set of processes that each contain at least one critical section, then the result of the computation performed may depend on their execution order.

**Definition (Mutual Exclusion)** At most one process is executing its critical section at any given time.

- Mutual exclusion is a solution to the CS problem.
- There are two main challenges that have to be taken into account when dealing with critical sections:
  - *Absence of Deadlocks*
    If some processes enter their critical sections, then one of them must eventually succeed.
  - *Absence of Starvation*
    If any process tries to enter its critical section, then that process must eventually succeed.
- In the following sections, the listing 2.22 is used as an example. This example also represents the common “critical section pattern”.

Listing 2.22: Critical Section Pattern in PROMELA

```
active proctype P() {
    do
        :: printf("P non-critical action\n");
           /* begin critical section */
           printf("P uses shared resources\n")
           /* end critical section */
    od
}

active proctype Q() {
    do
        :: printf("Q non-critical action\n");
           /* begin critical section */
           printf("Q uses shared resources\n")
           /* end critical section */
    od
}
```

## Mutual Exclusion

**First Attempt** Simple idea: Use boolean flags to control access to critical section, see listing **??**. But this does not yet solve guarantee mutual exclusion!

Listing 2.23: ME in PROMELA: First Attempt

```
bool enterCriticalP = false;
bool enterCriticalQ = false;

active proctype P() {
    do
        :: printf("P non-critical action\n");
           enterCriticalP = true;
           /* begin critical section */
           printf("P uses shared resources\n");
           /* end critical section */
           enterCriticalP = false
    od
}

active proctype Q() { /* analogous */ }
```

**Second Attempt: Busy Waiting** Busy waiting: Loop until the indicator variable gets a specific value, see listing **??**.

- Busy waiting works, but is problematic as it does not block the execution, even if the exclusion property is not fulfilled.
- This is wasteful on resources.

Listing 2.24: ME in PROMELA: Busy Waiting

```
bool enterCriticalP = false;
bool enterCriticalQ = false;

active proctype P() {
    do :: printf("P non-critical action\n");
          enterCriticalP = true;
          do :: !enterCriticalQ -> break
             :: else            -> skip
          od
          /* begin critical section */
          printf("P uses shared resources\n");
          /* end critical section */
          enterCriticalP = false
    od
}

active proctype Q() { /* analogous */ }
```

**Third Attempt: Blocking** Blocking: Use the exclusion property to release control if its not fulfilled and continue only when the exclusion properties are fulfilled. For this behavior, no assignment is used by the expression statement `!enterCriticalQ` which blocks when the expression evaluates to `false`. See listing **??** for an example implementation. This ensures ME, but is insufficient for proofing it using SPIN (see section **??** for more details).

Listing 2.25: ME in PROMELA: Busy Waiting

```
bool enterCriticalP = false;
bool enterCriticalQ = false;

active proctype P() {
    do :: printf("P non-critical action\n");
          enterCriticalP = true;
          !enterCriticalQ;  /* Block until enterCriticalQ is false. */
          /* begin critical section */
          printf("P uses shared resources\n");
          /* end critical section */
          enterCriticalP = false
    od
}

active proctype Q() { /* analogous */ }
```

### Variations

- Allow at most $n$ processes in critical sections.
  Modeling possibilities:
  - counters instead of booleans
  - semaphores
  - test-and-set instructions
- Refined mutual exclusion conditions
  - Several critical sections
  - Writers exclude each other and readers
    Readers exclude writers, but not other readers
    etc.
  - FIFO queues for entering sections (full semaphores)
- And many more...

## 2.5.9 Deadlocks

- *Invalid End State*
  - A process does not finish in an end state.
  - Okay, if it is not crucial to continue.
  - Two or more inter-dependent processes do not finish at the end $\Longrightarrow$ real deadlock.
- *Atomicity against Deadlocks*
  - More powerful and general mechanism than flags/blocking.
  - Often leads to conceptionally simpler solutions.
  - But is not always a realistic assumption!
- *Ideal Solution of ME Problem by Atomicity*
  - Check and set the critical section flag in one `atomic` step.

## 2.6 Distributed Systems

> "You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done." (Leslie Lamport)

Distributed systems consist of

- *nodes* that are connected by
- *communication channels* with
- *protocols* controlling the data flow among nodes.

Models of distributed systems try to abstract away from details of network/protocols/nodes. In PROMELA,

- nodes are modeled by processes,
- communication channels are modeled with PROMELA channels and
- protocols are modeled by algorithms distributed over the processes.

### 2.6.1 Channels

- The data type `chan` (short for channel) has two operations for sending and receiving data.
- A variable of type channel is declared with the initialization
  `chan name = [ capacity ] of { type1, ..., typeN }`
  - **`name`** The name of the channel variable.
  - **`capacity`** The capacity of a channel (how many values can be stored). Iff zero, the channel is a *rendezvous channel*. Otherwise a *buffered channel*. Must always be non-negative.
  - **`typeI`** PROMELA data types. Messages that are communicated via the channel are tuples $\in$ `type1` $\times \cdots \times$ `typeN`.
- Example: `chan ch = [2] of { mtype, byte, bool }`
  - Creates a channel, a reference to it is stored in `ch`.
  - Messages communicated via `ch` are triples $\in$ `mtype` $\times$ `byte` $\times$ `bool`
  - Given some `mtype = { red, yellow, green }`, an example message might be: `(green, 20, false)`
  - `ch` is a buffered channel, buffering up to 2 messages.

#### Sending/Receiving

The *send statement* has the form:

```
name ! expr1, ..., exprN
```

where

- `name` is the channel variable and
- `expr1, ..., exprN` is a sequence of expressions whose numbers and types have to match the declaration of the channel.
- This sends the values or `expr1, ..., exprN` as a single message over the channel.
- Example: `ch ! green, 20, false`

The *receive statement* has the form:

```
name ? var1, ..., varN
```

where

- `name` is the channel variable and
- `var1, ..., varN` is a sequence of variables whose numbers and types have to match the declaration of the channel.
- This assigns the values of a single message to the variables `var1, ..., varN` and removes the message from the channel.
- Example: `ch ? color, time, flash`
- An alternative syntax is `name ? var1(var2, ..., varN)`.
- For copying a message without removing it, the variable names have to be wrapped in `< >`, e.g. `ch ? <color, time, flash>`

The executability table for channels is shown in table 2.2.

| Statement Type | Executable |
|---|---|
| `name ! msg` (rendezvous) | iff some process wants to receive from the channel |
| `name ? msg` (rendezvous) | iff a message is available in the channel |
| `name ! msg` (buffered) | iff the message queue is not full, i.e. $n <$ capacity |
| `name ? msg` (buffered) | iff the channel is not empty, i.e. $n > 0$ |

Table 2.2: Executability of Send/Receive in PROMELA

**Pattern Matching** It is also possible to pass values to the receive command:

```
ch ? exp1, ..., expN
```

- Each expression is either a variable or a value and the types of all expressions must comply to the type of `ch`.
- Each expression `expI` is then matched against the message `msgI` returned from `ch`:
  - If `expI` is a value, then `expI = msgI` must hold.
  - If `expI` is a variable, then `msgI` is assigned to `expI`.
  - Otherwise, matching fails.
- The receive statement is only executable iff matching succeeds.

**Random Receive** For a buffered channel, the syntax

```
ch ?? exp1, ..., expN
```

can be used. It is executable iff a matching message exists somewhere in the channel, not necessarily in the front of the queue.

- If executed, the first matching message is removed from the channel.
- Can be used to transmit messages with different purposes over one channel.
- The name “random receive” is confusing - the matching is executed deterministically!

## Scope

- *Global channel*
  - Standard case, channels are typically declared globally.
  - All processes can send/receive messages.
- *Local channel*
  - Less often used.
  - Dies with its process.
  - Can be useful for modeling security issues.
  - A reference to a local channel may be passed through a global channel.

## Rendezvous

Because of the executability rules of rendezvous channels, the transfer of a message from sender to receiver is synchronous, i.e. in a single operation. There are only two possibility for the execution of send/receive through a rendezvous channel:

1. The sender arrives first.
   a) The location counter of the sender process is at send (“offer to engage in rendezvous”).
   b) The location counter of the receiver process is at receive (“rendezvous can be accepted”).
2. The receiver arrives first.
   a) The location counter of the receiver process is at receive (“offer to engage in rendezvous”).
   b) The location counter of the sender process is at send (“rendezvous can be accepted”).

In either case, the location counter both processes is incremented at once. This is the only place in PROMELA where processes are executed synchronously!

**Drawbacks** Rendezvous are too restrictive for many application.

- Servers and clients block each other too much.
- It is difficult to manage uneven workload (small number of servers service thousands of clients).
- Make the channel types as small as possible (this also holds for buffered channels).

### Buffered

Buffered channels queue messages, so that requests/services do not immediately block clients/servers. Buffered channels with capacity `cap`...

- ...can hold up to `cap` messages.
- ...are FIFO (first-in-first-out) data structures: the oldest message is retrieved by receive.
- The receive statements by default reads and removes a message.
- Sending/receiving from/to buffered channels is asynchronous, so interleaving may occur between sending and receiving.
- The executability is shown in table 2.2.
- With the SPIN option `-m` it is possible to prescribe a different sending semantics: send to a full channel does not block, but the message is lost instead.

**Checking Channels for being Full/Empty** In order to prevent blocking, it is possible to check the state of a channel `ch`:

- `full(ch)` Checks whether `ch` is full.
- `nfull(ch)` Checks whether `ch` is not full.
- `empty(ch)` Checks whether `ch` is empty.
- `nempty(ch)` Checks whether `ch` is not empty.

These guards cannot be negated and thus shall not be combined with `else` (this is an implicit negation), which can result in unintuitive blocking behavior.

**Drawbacks** Buffered channels are part of the state which enlarges the state space that has to be traversed during verification a lot!

- Do not use buffered channels unless they are needed.
- Set the capacity as low as possible.
- Make the channel types as small as possible (this also holds for rendezvous channels).

## 2.6.2 Client-Server Model

Listing 2.26: Client-Server Model in PROMELA

```
1 chan request = [0] of { byte };
2
3 active proctype Server() {
4     byte num;
5     do :: request ? num ->
6           printf("Serving client %d\n", num)
7     od
8 }
```

```
9
10 active proctype Client0() {
11     request ! 0
12 }
13
14 active proctype Client1() {
15     request ! 1
16 }
```

The basic code for the client-server Model is shown in listing 2.26. Because of interleaving, the order of sending is non-deterministic. The execution

```
> spin  client-server.pml
      Serving client 1
      Serving client 0
      timeout
#processes: 1
  5:    proc  0 (Server:1) code/promela/client-server.pml:5 (state 3)
3 processes created
```

produced a timeout in the server as not further messages are sent by the clients and the server blocks forever.

### Reply Channels

Listing 2.27: Client-Server Model with Replys in PROMELA

```
1  mtype = { nice, rude };
2
3  chan request = [0] of { mtype };
4  chan reply   = [0] of { mtype };
5
6  active [2] proctype Server() {
7      mtype msg;
8      do :: request ? msg -> reply ! msg
9      od
10 }
11
12 active proctype ClientNice() {
13     mtype msg;
14     request ! nice;  reply ? msg;
15     assert(msg == nice);
16 }
17
18 active proctype ClientRude() {
19     mtype msg;
20     request ! rude;  reply ? msg;
21     assert(msg == rude);
22 }
```

The above client-server model does not yet have replies which is not very useful. A second channel can be used to reply for the clients. The listing 2.27 shows a slightly different example where the client sends a message and the server is asked to respond with the same message.

This protocol has the problem that interleaving may occur and the wrong response is sent (if more than one server is used).

### Sending Channels via Channels

Listing 2.28: Client-Server Model with Sending Channels in PROMELA

```
1  mtype = { nice, rude };
2
3  chan request = [0] of { mtype, chan };
4
5  active [2] proctype Server() {
```

```
6       mtype msg;  chan ch;
7       do :: request ? msg, ch -> ch ! msg
8       od
9   }
10
11  active proctype ClientNice() {
12      chan reply = [0] of { mtype };
13      mtype msg;
14      request ! nice, reply;  reply ? msg;
15      assert(msg == nice);
16  }
17
18  active proctype ClientRude() {
19      chan reply = [0] of { mtype };
20      mtype msg;
21      request ! rude, reply;  reply ? msg;
22      assert(msg == rude);
23  }
```

One way to fix the protocol is that the clients declare local reply channels that are then send to the server, as shown in listing 2.28.

### Sending Process IDs

The above examples used fixed constants for client identification (`nice`, `rude`). This is inflexible, produces brittle code and does not scale for lots of clients. An improvement is to use process IDs, `_pid`, for identification. The code in listing 2.29 shows the client code for that process.

Listing 2.29: Client-Server Model with PID in PROMELA

```
1  byte serverID, clientID;
2  chan reply = [0] of { byte, byte };
3  request ! reply, _pid;
4  reply ? serverID, clientID;
5
6  assert(clientID == _pid)
```

# 3 Linear Temporal Logic (LTL)

## 3.1 Propositional Logic

### 3.1.1 Syntax

**Definition (Signature of Propositional Logic)** A set of *propositional variables* (with typical elements $p$, $q$, $r$, ...).

**Definition (Propositional Formulas)** The set of propositional formulas $\mathit{For}_0$ is inductively defined as

- The truth constants **true**, **false** and the variables $\mathcal{P}$ are formulas.
- If $\phi$ and $\psi$ are formulas, then all of the following are also formulas:

$$\neg\phi, \quad (\phi \wedge \psi), \quad (\phi \vee \psi), \quad (\phi \rightarrow \psi), \quad (\phi \leftrightarrow \psi)$$

  The formulas are connected using the *propositional connectives*, listed in table 3.1.

### 3.1.2 Semantics

**Definition (Propositional Interpretation)** An interpretation $\mathcal{I}$ assigns a truth value to each propositional variable:

$$\mathcal{I} : \mathcal{P} \rightarrow \{\mathfrak{t}, \mathfrak{f}\}$$

**Definition (Valuation Function)** The valuation semantic under an interpretation $\mathcal{I}$ is defined by the valuation function

$$\mathrm{val}_{\mathcal{I}} : \mathit{For}_0 \rightarrow \{\mathfrak{t}, \mathfrak{f}\}$$

| Name | Symbol |
|---|---|
| Negation | $\neg$ |
| Conjunction (“and”) | $\wedge$ |
| Disjunction (“or”) | $\vee$ |
| Implication | $\rightarrow$ |
| Equivalence | $\leftrightarrow$ |

Table 3.1: Propositional Connectives

with the following semantics (let $p \in \mathcal{P}$, $\phi, \psi \in \mathit{For}_0$):

$$
\begin{aligned}
\mathrm{val}_{\mathcal{I}}(p) &:= \mathcal{I}(p) \\
\mathrm{val}_{\mathcal{I}}(\textbf{true}) &:= \mathfrak{t} \\
\mathrm{val}_{\mathcal{I}}(\textbf{false}) &:= \mathfrak{f} \\
\mathrm{val}_{\mathcal{I}}(\neg\phi) &:= \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{\mathcal{I}}(\phi) = \mathfrak{f} \\ \mathfrak{f} & \text{otherwise} \end{cases} \\
\mathrm{val}_{\mathcal{I}}(\phi \wedge \psi) &:= \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{\mathcal{I}}(\phi) = \mathfrak{t} \text{ and } \mathrm{val}_{\mathcal{I}}(\psi) = \mathfrak{t} \\ \mathfrak{f} & \text{otherwise} \end{cases} \\
\mathrm{val}_{\mathcal{I}}(\phi \wedge \psi) &:= \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{\mathcal{I}}(\phi) = \mathfrak{t} \text{ or } \mathrm{val}_{\mathcal{I}}(\psi) = \mathfrak{t} \\ \mathfrak{f} & \text{otherwise} \end{cases} \\
\mathrm{val}_{\mathcal{I}}(\phi \to \psi) &:= \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{\mathcal{I}}(\phi) = \mathfrak{f} \text{ or } \mathrm{val}_{\mathcal{I}}(\psi) = \mathfrak{t} \\ \mathfrak{f} & \text{otherwise} \end{cases} \\
\mathrm{val}_{\mathcal{I}}(\phi \leftrightarrow \psi) &:= \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{\mathcal{I}}(\phi) = \mathrm{val}_{\mathcal{I}}(\psi) \\ \mathfrak{f} & \text{otherwise} \end{cases}
\end{aligned}
$$

### 3.1.3 Semantic Notions

Let $\phi \in \mathit{For}_0$, $\Gamma \subseteq \mathit{For}_0$.

**Definition (Satisfying Interpretation, Consequence Relation)** An interpretation $\mathcal{I}$ *satisfies* $\phi$ (write $\mathcal{I} \models \phi$) iff $\mathrm{val}_{\mathcal{I}}(\phi) = \mathfrak{t}$.

$\phi$ *follows from* $\Gamma$ (write $\Gamma \models \phi$) iff for all interpretations $\mathcal{I}$:

$$\text{If } \mathcal{I} \models \psi \text{ for all } \psi \in \Gamma \text{ then also } \mathcal{I} \models \phi$$

which is equivalent to

$$\{\, \mathcal{I} : \mathcal{I} \models \psi \text{ for all } \psi \in \Gamma \,\} \subseteq \{\, \mathcal{I} : \mathcal{I} \models \phi \,\}$$

**Definition (Satisfiability, Validity)** A formula is *satisfiable* if it is satisfied by some interpretation. If every interpretation satisfied $\phi$ (write $\models \phi$), then $\phi$ is called *valid*.

### 3.1.4 Expressiveness

- Propositional logic can only express properties for one state of a program.
- It is needed to express properties over every run of a program, including state changes and so on.
- This cannot be done by propositional logic $\implies$ Linear temporal logic.

## 3.2 Kripke Structure and Transition Systems

- Each program state $s_j$ has its own propositional interpretation $\mathcal{I}_j$. The values of the variables are listen in lexicographic order below the “state name”, see figure 3.1 for an example.

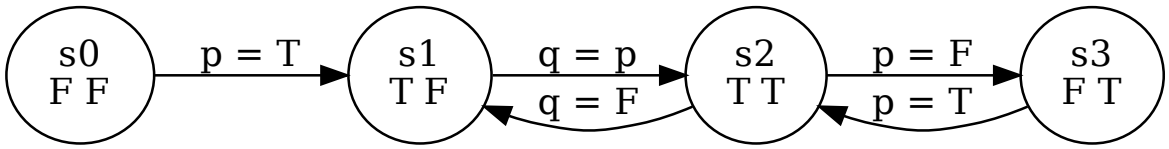


Figure 3.1: Kripke Structure

| Name | Symbol | Intution |
|---|---|---|
| Always | $\Box\phi$ | $\phi$ has to hold on the entire subsequent run |
| Sometimes | $\Diamond\phi$ | $\phi$ eventually has to hold (somewhere) |
| Until | $\phi\mathcal{U}\psi$ | $\psi$ at the current or a future position and $\phi$ has to hold until that position |

Table 3.2: Temporal Connectives

- Computations (or "runs") are infinite paths through states (infinite is not a restriction! Let a finite run be stuck at the final state).
- In general, infinitely many different runs are possible.
- Temporal logic is about expressing properties like that a variables changes its value infinitely many often in each run.

## 3.3 Syntax

**Definition (Syntax of Temporal Logic)** Temporal logic is an extension of the propositional logic with the three following formulas. Let $\phi$ and $\psi$ be formulas, then all of the following are also formulas:

$$\Box\psi, \quad \Diamond\psi, \quad \psi\mathcal{U}\phi$$

The formulas are connected using the *temporal connectives*, listed in table 3.2.

## 3.4 Semantics

Temporal logic semantics generalize the semantics of propositional logic relative to a sequence of interpretations rather than relative to one interpretation.

**Definition (Run and Suffix)** A *run* $\sigma$ of a transition system is an infinite chain of states $s_j$ where $\mathcal{I}_j$ is the propositional interpretation of variables in the $j$-th state. Write $\sigma = s_0 s_1 \cdots$.

Let $\sigma = s_0 s_1 \cdots$ be a run, then $\sigma|_i$ denotes the *suffix* $s_i s_{i+1} \cdots$ of $\sigma$.

**Definition (Validity Relation)** The validity of temporal formulas depends on runs $\sigma = s_0 s_1 \cdots$. Let $p \in \mathcal{P}$ and $\psi, \phi \in \mathit{For}_0$, then the validity relation is defined as:

$$
\begin{array}{lcl}
\sigma \models p & \iff & \mathcal{I}_0(p) = \mathrm{t} \\
\sigma \models \neg\phi & \iff & \text{not } \sigma \models \phi \text{ (write } \sigma \not\models \phi \\
\sigma \models \phi \wedge \psi & \iff & \sigma \models \phi \text{ and } \sigma \models \psi \\
\sigma \models \phi \vee \psi & \iff & \sigma \models \phi \text{ or } \sigma \models \psi \\
\sigma \models \phi \rightarrow \psi & \iff & \sigma \not\models \psi \text{ or } \sigma \models \psi \\
\sigma \models \Box\phi & \iff & \sigma|_k \models \phi \text{ for all } k \geq 0 \\
\sigma \models \Diamond\phi & \iff & \sigma|_k \models \phi \text{ for some } k \geq 0 \\
\sigma \models \phi \mathcal{U} \psi & \iff & \sigma|_k \models \psi \text{ for some } k \geq 0 \text{ and } \sigma|_j \models \phi \text{ for all } 0 \leq j < k
\end{array}
$$

So propositional formulas are always evaluated in the initial state of some suffix or run $\sigma$.

### 3.4.1 Validity

**Definition (Validity)** A formula $\phi$ is called *valid* (write $\models \phi$) iff for all runs $\sigma$ it holds that $\sigma \models \phi$.

**Definition (Representation of Runs)** A set of runs can be represented as a sequence of propositional formulas: $\phi_0 \phi_1 \cdots$ represents all runs $s_0 s_1 \cdots$ such that $\mathcal{I}_j \models \phi_j$ for $j \geq 0$.

### 3.4.2 Transition Systems

**Definition (Transition System)** A *transition system* $\mathcal{T} = (S, \mathrm{Ini}, \delta, \mathcal{I})$ is composed of:

- Set of states $S$,
- set $\mathrm{Ini} \subseteq S$, $\mathrm{Ini} \neq \emptyset$ of initial states,
- transition relation $\delta \subseteq S \times S$ and
- labeling $\mathcal{I}$ of each state $s \in S$ with a propositional interpretation $\mathcal{I}_s$.

**Definition (Run of Transition System)** A *run* of $\mathcal{T}$ is a sequence of states $\sigma = s_0 s_1 \cdots$ so that $s_0 \in \mathrm{Ini}$ and $(s_i, s_{i+1} \in \delta$ for all $i$.

**Definition (Validity Relation)** Given a transition system $\mathcal{T}$, a temporal formula $\phi$ is *valid* $\mathcal{T}$ (write $\mathcal{T} \models \phi$) iff $\sigma \models \phi$ for all runs $\sigma$ of $\mathcal{T}$.

## 3.5 Safety and Liveness Properties

### 3.5.1 Safety Properties

- Always-formulas called *safety property*: “something bad never happens”
- Let `mutex` be a variable that is true when two processes do not access a critical resource at the same time (so true is good).
- The formula $\Box$`mutex` expresses that simultaneous access never happens.

### 3.5.2 Liveness Properties

- Sometimes-formulas called *liveness properties*: "something good happens eventually"
- Let s be a variable that is true when a process delivers a service.
- The formula $\Diamond$s expresses that the service is eventually provided.

### 3.5.3 Complex Properties

$$\Box\Diamond\phi$$

A property $\phi$ becomes true infinitely often during a run (but not necessarily successive or maybe with large gaps).

$$\Diamond\Box\phi$$

Once in time, a property $\phi$ becomes true forever (but not necessarily is at the start).

## 3.6 Formal Languages

**Definition (Formal Languages)** Given a finite alphabet $\Sigma$, a word $w \in \Sigma^*$ is a finite sequence $w = a_0 \cdots a_n$ with $a_i \in \Sigma$, $i \in \{ 0, \cdots, n \}$. The subset $\mathcal{L} \subseteq \Sigma^*$ is called a *language* over $\Sigma$.

**Definition (Formal $\omega$-Languages)** Given a finite alphabet $\Sigma$, an $\omega$-word $w \in \Sigma^\omega$ is an infinite sequence $w = a_0 \cdots a_k \cdots$ with $a_i \in \Sigma$, $i \in \mathbb{N}$. The subset $\mathcal{L} \subseteq \Sigma^\omega$ is called an $\omega$-language over $\Sigma$.

### 3.6.1 Büchi Automaton

**Definition (Büchi Automaton)** A (non-deterministic) Büchi automaton over an alphabet $\Sigma$ consists of a

- finite, non-empty set of *locations* (or *states*) $Q$,
- non-empty set of *initial/start* locations $I \subseteq Q$,
- set of *accepting/final* locations $F = \{ F_1, \cdots, F_n \} \subseteq Q$ and
- transition relation $\delta \subseteq Q \times \Sigma \times Q$.

**Definition (Run and Accepted Run)** An infinite word $w = a_0 \cdots a_k \cdots \in \Sigma^\omega$ is a *run* of a Büchi automaton if $q_{i+1} \in \delta(q_i, a_i)$ for all $i \geq 0$ and some initial location $q_0 \in I$.

A Büchi automaton *accepts* a run $w \in \Sigma^\omega$, if some accepting location $f \in F$ is infinitely often visited during $w$.

#### $\omega$-Regular Expressions

$\omega$-regular expressions are like standard regular expressions but with one additional rule, see table 3.3.

| | |
|---|---|
| $ab$ | $a$ then $b$ |
| $a + b$ | $a$ or $b$ |
| $a^*$ | arbitrarily, but finitely often $a$ |
| $a^\omega$ | infinitely often $a$ |

Table 3.3: $\omega$-Regular Expressions

**LTL and Büchi Automaton**

- LTL and Büchi automata are connected.
- A run of the transition system is an infinite sequence of interpretations $\mathcal{I}$.
- Given an LTL formula $\phi$, a Büchi automaton can be constructed accepting exactly those runs (infinite sequences of interpretations) that satisfy $\phi$.
- Given a set $\mathcal{P}$ of propositional variables, the alphabet $\Sigma$ of the Büchi automaton has to contain all interpretations over $\mathcal{P}$, i.e. $\Sigma = 2^{\mathcal{P}}$ (the power set of $\mathcal{P}$).
- Sadly, the transitions $\delta$ have to be constructed manually...

## 3.7 Decidability, Closure Properties

Many properties for regular finite automata also hold for Büchi automata while the latter being strictly more expressive (deterministic automata cannot accept all $\omega$-regular expressions).

**Theorem** DecidabilityIt is decidable whether the accepted Language $\mathcal{L}^\omega(\mathcal{B})$ of a Büchi automaton is empty.

**Theorem** Closure Properties The set of $\omega$-regular languages is closed w.r.t. intersection, union and complement:

- If $\mathcal{L}_1$, $\mathcal{L}_2$ are $\omega$-regular, then $\mathcal{L}_1 \cap \mathcal{L}_2$ and $\mathcal{L}_1 \cup \mathcal{L}_2$ are also $\omega$-regular.
- If $\mathcal{L}$ is regular then $\Sigma^\omega \setminus \mathcal{L}$ is also $\omega$-regular.

# 4 SPIN

- The main goal of SPIN model checking is to fight design flaws by offering a model-based methodology for improving the design and to exhibit the defects of concurrent and distributed software systems.
- Some problems of concurrent/distributed systems:
  - Hard to predict and hard to form a correct intuition about them.
  - Enormous combinatorial explosion of possible behaviors.
  - Interleaving of processes is prone to perform unsafe operations (“data races”).
  - Countermeasures are prone to cause deadlocks.
  - The application itself has limited control possibilities over external factors:
    * Scheduling strategies,
    * (relative) speed of components,
    * performance and reliability of communication mediums.
- Testing concurrent/distributed systems is also hard, as they cannot be testes exhaustively:
  - Lack of controllability (scheduling, delays, . . . )
    Some failures are missed in the test phase.
  - Lack of reproducibility
    Even if a failure appears, it is often impossible to analyze/debug it.
  - Lack of resources
    Exhaustive testing exhausts the tester long before it exhausts the behavior of the system.
- The main challenges of modeling are the conflicting goals
  - *richness*, a model must be rich enough to encompass defects the real system could have and
  - *simplicity*, a model must be simple enough to be checkable, both theoretically and in practice.
- Verify the property the system should have, not a property the system already has!

#### The general usage scenario for PROMELA

1. Model the essential features of a system in PROMELA.
   - Abstract away from complex (numerical) computations (make use of non-deterministic choices of the outcome).
   - Replace unbounded data structures with fixed approximations.
   - Assume a fair process scheduler.
2. Identify and select the properties that the model must satisfy.

- Mutual exclusion
- Absence of deadlock
- Absence of starvation
- Event sequences (e.g. system responsiveness)

3. Verify that all possible runs satisfy these properties.
   - Typically several iterations are needed to get the model and the system properties right.
   - Failed verification attempts provide feedback in the form of counter examples.

## 4.1 Model Checking

- A model checker is designed to proof the designer wrong. Thus it does not try to proof correctness, but the opposite!
- Tuned to find counter examples to a correctness property.
- On the other hand, this can lead to proofing correctness: The absence of counter examples proofs that the validated properties are correct.
- The counter example search is exhaustive. That is, non-determinism is resolved in every possible way.

### 4.1.1 SPIN

- SPIN, an abbreviation for "Simple Promela Interpreter", can simulate a model (randomly/interactively/guided) and can generate a verifier.
- This generated verifier is a C program that than performs the model checking:
  - Exhaustively checks the PROMELA model against correctness properties.
  - In the case the check is negative: Generates a failing run of the model that can be simulated by SPIN.

### 4.1.2 LTL Properties

Given a transition system $\mathcal{T}$ (e.g. derived from a PROMELA program), the verification task is to check whether an LTL formula $\phi$ is satisfied in all runs of $\mathcal{T}$, i.e. $\mathcal{T} \models \phi$. The process has the following steps steps:

1. Represent the transition system $\mathcal{T}$ as a Büchi automaton $\mathcal{B}_{\mathcal{T}}$ such that $\mathcal{B}_{\mathcal{T}}$ accepts exactly those runs corresponding to runs through $\mathcal{T}$.
2. Construct a Büchi automaton $\mathcal{B}_{\neg\phi}$ for the negation of formula $\phi$.
3. If $\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi}) = \emptyset$, then $\phi$ holds. Otherwise a counterexample was found. To calculate $\mathcal{L}^{\omega}(\mathcal{B}_{\mathcal{T}}) \cap \mathcal{L}^{\omega}(\mathcal{B}_{\neg\phi})$, construct an intersection automaton and search for cycles through accepting states.

## 4.2 Workflow

### 4.2.1 Guided Simulation

### 4.2.2 Temporal Properties

## 4.3 Fairness

**Definition (Weak Fairness)** A run is *weakly fair* iff the following holds: Each continuously executable statement is executed eventually.

## 4.4 Correctness Properties

Given a PROMELA model $M$ and correctness properties $C_1, \cdots, C_n$ and assume there is a check whether a PROMELA run $R$ satisfies property $C$.

**Definition (Correctness of model relative to property)** Let $R_M$ be the set of all possible runs of a PROMELA model $M$. Then

- For each correctness property $C_i$, $R_{M,C_i}$ is the set of all runs of $M$ satisfying $C_i$ (clearly $R_{M,C_i} \subseteq R_M$).
- $M$ is correct relative to $C_1, \cdots, C_n$ iff $R_M = (R_{M,C_1} \cap \cdots \cap R_{M,C_n})$.
- If $M$ is not correct, then each $r \in \left(R_M \setminus (R_{M,C_1} \cap \cdots \cap R_{M,C_n})\right)$ is a counter example.

These correctness properties can be formulated within or outside of a PROMELA model:

- Stating properties *within* of the model.
  - Assertion statements (section 4.4.1)
  - Meta labels
    - `end` labels (section 4.4.2)
    - `accept` labels (section 4.4.3)
    - `progress` labels (not covered)
- Stating properties *outside* of the model.
  - Never claims (section 4.4.4)
  - Temporal logic formulas (section 4.4.4)

### 4.4.1 Assertions

**Definition (Assertion Statement)** Assertion statements in PROMELA have the form/syntax

```
assert(expr)
```

where `expr` is any PROMELA expression.

- Typically, but not necessarily, the expression evaluates to a boolean value (type `bool`).
- The assert statement can take any statement position.
- Effects (in both simulation and model checking mode):
  - No effect iff the expression evaluates to some non-true value.
  - Triggers an error iff the expression evaluates to zero.
- The workflow for performing model checking is described in section 4.2.
- Notice that assertions cannot express full function verification as quantification is not possible. But they can perform sanity checks/partial verification.

### 4.4.2 Valid End States

- A blocking process is legal, as long as other processes can proceed.
- Blocking for letting other processes proceed is useful and required, e.g. for concurrent and distributed models.
- But it is an error if a process blocks while no other process can proceed $\implies$ Deadlock.

**Definition (Valid End State)** An end state of a run is valid iff the location counter of each process is at an end location.

**Definition (End Location)** *End locations* of a process $P$ are:

- The textual end of $P$ (i.e. the end of the process, the last closing brace).
- Any location marked with an *end label* of the form `endXXX:`.

### 4.4.3 Acceptance Labels/Cycles

**Definition (Acceptance Location)** A location that is marked with an *acceptance label* of the form `acceptXXX:` is called an *accept location*.

**Definition (Acceptance Cycle)** A run that passes an acceptance label infinitely often is called an *acceptance cycle*.

Acceptance cycles are mainly used in never claims (see 4.4.4) to define forbidden infinite behavior.

### 4.4.4 Temporal Properties

- Lots of correctness properties are not expressible by assertions:
  - All properties involving state changes.
  - Properties that depend on the scheduling of two processes.
- Temporal logic can be used to increase the expressiveness of specifications.
- Notice: Even temporal logic is not expressive enough to cover all properties of a PROMELA model, but in practice it is often sufficient.

### Drawbacks of Assertions

- They only specify about the state at their location in the code.
- No separation of concerns (model vs. correctness property, the model is mixed with the validation).
- Changing assertions is prone to cause errors, especially if they were copied to lots of locations.
- It is easy to forget assertions, some correctness properties might be violated at unexpected locations.
- At the biggest drawback: Many properties are not expressible via assertions!

## Boolean Temporal Logic

- LTL only knows propositional formulas including only propositional variables.
- But expressions like `0 <= i <= len - 1` contain numeric variables.
- So a slight generalization of LTL is required.
- In *Boolean Temporal Logic*, the atomic building blocks are boolean expressions over PROMELA variables rather than propositional variables.

**Definition (Boolean Temporal Logic Formulas)** The set of *boolean temporal formulas* $For_{BTL}$ is inductively defined as

- All global PROMELA variables and constants of type `bool`/`bit` are formulas.
- If `e1` and `e2` are numerical PROMELA expressions, then all the following are formulas:

  `e1 == e2`, `e1 != e2`, `e1 < e2`, `e1 <= e2`, `e1 > e2`, `e1 >= e2`
- If `P` is a process and `l` is a label in `P`, then `P@l` is a formula (read as "`P` is at `l`").
- If $\phi$ and $\psi$ are formulas, then all the following are formulas:

  `!`$\phi$, $\phi$ `&&` $\psi$, $\phi$ `||` $\psi$, $\phi$ `->` $\psi$, $\phi$ `<->` $\psi$, `[]`$\phi$, `<>`$\phi$, $\phi$`U`$\psi$

  where the latter three represent the LTL formulas $\Box\phi$, $\Diamond\phi$ and $\phi\mathcal{U}\psi$.

**Semantics** A run $\sigma$ through a PROMELA model $M$ is a chain of states $s_1 s_2 \cdots$ with the state tuple $s_i = (\mathcal{L}_i, \mathcal{I}_i)$:

- $\mathcal{L}_i$ maps each running process to its current location counter.
- From $\mathcal{L}_i$ to $\mathcal{L}_{i+1}$, only one of the location counters can advance (exception: channel rendezvous).
- $\mathcal{I}_i$ maps each variable in $M$ to its current (possible non-boolean) value.

Arithmetic and relation expressions are then interpreted in the states as expected, e.g. $\mathcal{L}_j, \mathcal{I}_j \models$ `x < y` iff $\mathcal{I}_j(\texttt{x}) < \mathcal{I}_j(\texttt{y})$. Label expressions are evaluated as: $\mathcal{L}_j, \mathcal{I}_j \models$ `Pl` iff $\mathcal{L}_j(\texttt{P})$ is at the location labeled with `l`.

## Model Checking a Temporal Property

The general form for the definition of a TL formula is:

```
ltl <claim name> { <TL formula> }
```

where

**<claim name>** is the name of the claim (if more than one formula is defined).

**<TL formula>** the formula itself.

It is possible to define more than one LTL formula in the PROMELA file, but it is not possible to verify more than one LTL formula at once!

## Never Claims

- Disproving $\Diamond\phi$ needs a representation of an infinite computation.
  1. Negated TL formula is translated to specific PROMELA process.
  2. That so-called "never claim" process tries to show the user is wrong.
  3. A Büchi automaton is derived from the "never claim" process.
- Never claims are translated into special a PROMELA process with accept labels.
  - Of one of those is reached infinitely often, the claim is disproved.
  - Can be viewed as a game between SPIN and the user:
    * SPIN makes a forward move in the never claim process with accept labels.
    * The user makes a forward move in the PROMELA program under verification.
    * If an accept label is reached infinitely many often, the user lost and verification fails.
  - Finiteness is guaranteed by cyclicity check.

Listing 4.1: Never Claim PROMELA Process

```
never {  /* !(<>q) */
    accept_init:
    T0_init:
    if
    :: (!q) -> goto T0_init
    fi
}
```

Given some PROMELA program $P$ over a variable $q$ and the attempt to prove $P \models \Diamond q$, a never claim process similar to listing 4.1 is generated. The following is possible:

- If $q$ becomes true, SPIN is blocked at that point and "loses".
- If $q$ never becomes true, SPIN loops infinitely often over the accept label resulting in an acceptance cycle.

PROMELA has a specific construct only for never claim process. Some never claims are not expressible in temporal logic!

## Liveness Properties

## Label Checking

- With label checking it is possible to verify things like mutual exclusion without the usage of ghost variables.
- E.g. all critical sections can be marked with the label `cs:` and be verified with an LTL formula `[]!(Pcs && Qcs)` (`P` and `Q` are never in the CS at the same time).
- Labels expressions can often remove the need for ghost variables, not only for ME validation.

# 5 First-Order Logic (FOL)

- Java programs will be specified with the Java Modeling Language (JML), see chapter 6.
- JML combines Java expressions and First-Order Logic (FOL).
- Java programs will be verified using Dynamic Logic (DL), see chapter 7.
- DL combined First-Order Logic (FOL) and Java programs,

## 5.1 Syntax

**Definition (First-Order Signature)** The first-order *signature* $\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym})$ over a set $\mathrm{TSym}$ of types consists of:

- A set $\mathrm{FSym}$ of typed function symbols, declared $f : A_1 \times \cdots \times A_n \to A$ with $n$ argument types $A_i \in \mathrm{TSym}$ and return type $A \in \mathrm{TSym}$.
- A set $\mathrm{PSym}$ of typed predicate symbols, declared $p(A_1, \cdots, A_n)$ with $n$ argument types $A_i \in \mathrm{TSym}$.
- A set $\mathrm{VSym}$ of typed variable symbols, declared $v : A$ with type $A \in \mathrm{TSym}$.

**Definition (Constants)** A function symbol $f \in \mathrm{FSym}$ with $0$ arguments is called a *constant*.

**Definition (Propositional Variable)** A predicate symbol $p \in \mathrm{PSym}$ with $0$ arguments is called a *propositional variable*.

### 5.1.1 Terms

**Definition (First-Order Terms)** The set of first-order terms $\mathrm{Trm}_A$ for a type $A$ is inductively defined as

- The variable symbols $\mathrm{VSym}$ of type $A$ are terms.
- The function $f(t_1, \cdots, t_n)$ where $t \in \mathrm{FSym}, t : A_1 \times \cdots \times A_n \to A$ and each $t_i$ is a term of type $A_i$.

First-order terms are like side effect-free expressions in programs.

### 5.1.2 Formulas

**Definition (First-Order Formulas)** The set of first-order formulas $\mathrm{Fml}$ is inductively defined as

- All of the following *atomic formulas* are formulas:
  - **true**

  - **false**
  - $t_1 \doteq t_2$ (“equality”)
  - $p(t_1, \cdots, t_n)$ (“predicate”) where $p \in \mathrm{PSym}$ is declared $p(A_1, \cdots, A_n)$ and each $t_i$ is a term of type $A_i$.
- If $\phi$ and $\psi$ are first-order formulas and $x$ a variable of type $A$, then all of the following are also formulas:

$$\neg\phi, \quad \phi \wedge \psi, \quad \phi \vee \psi, \quad \phi \to \psi, \quad \phi \leftrightarrow \psi, \quad \forall x; \phi, \quad \exists x; \phi$$

  In $\forall x; \phi$ and $\exists x; \phi$, the variable $x$ occurs *bound* (i.e. *not free*). Formulas with no free variables are called *closed*.

## 5.2 Semantics

- In propositional logic, an interpretation of variables with $\{ \mathfrak{t}, \mathfrak{f} \}$ works out.
- In first-order logic a meaning must be assigned to:
  - Variables bound in quantifiers.
  - Constant and function symbols.
  - Predicate symbols.
- Each variable or function may denote a different object.
- Typing must be respected: Different types must denote different objects.
- A first-order formula must be interpreted relative to a signature $\Sigma$:
  1. A types universe (domain) of objects.
  2. A mapping from variables to objects of suitable types.
  3. A mapping from function arguments to function values.
  4. The set of argument tuples where a predicate is true.

#### 1: First-Order Domains/Universes

**Definition (First-Order Universe/Domain)** A non-empty set $\mathcal{D}$ is a *universe* or *domain*. Each element of $\mathcal{D}$ has a fixed given by $\delta : \mathcal{D} \to \mathrm{TSym}$.

- Notation for the domain elements of type $A \in \mathrm{TSym}$: $\mathcal{D}^A = \{ d \in \mathcal{D} : \delta(d) = A \}$
- Obviously, $\mathcal{D} = \bigcup_{A \in \mathrm{TSym}} \mathcal{D}^A$ and if $A \neq B$ then $\mathcal{D}^A \cap \mathcal{D}^B = \emptyset$.
- Each type must “contain” at least one domain element, i.e. $\mathcal{D}^A \neq \emptyset$ for all $A \in \mathrm{TSym}$.

Warning:

- Domain elements differ from the terms representing them.
- First-order formulas and terms do not have access to the domain!
  - $\mathcal{D}$, $\delta$, $\mathcal{D}^{\mathcal{T}}$ are not part of the FOL signature.
  - Compare to language (e.g. Java) without access to the heap representation.

**2: Variable Assignments** Think of variable assignments as an environment for storage of local variables.

**Definition (Variable Assignment)** A *variable assignment* $\beta : \mathrm{VSym} \to \mathcal{D}$ maps variables to domain elements while respecting the variable type. That is, if $x$ has type $A$ then $\beta(x) \in \mathcal{D}^A$.

**Definition (Modified Variable Assignment)** Let $y$ be a variable of type $A$, $\beta$ a variable assignment and $d \in \mathcal{D}^A$. Then the *modified variable assignment* is:

$$\beta_y^d(x) \coloneqq \begin{cases} \beta(x) & \text{iff } x \neq y \\ d & \text{iff } x = y \end{cases}$$

**3/4: First-Order States/Models**

**Definition (First-Order State/Model)** Let $\mathcal{D}$ be a domain with the typing function $\delta$.

- If $f \in \mathrm{FSym}$ is declared as $f : A_1 \times \cdots \times A_n \to A$, then an interpretation of $f$ is a function $\mathcal{I}(f) : \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n} \to \mathcal{D}^A$.
- If $p \in \mathrm{PSym}$ is declared as $p(A_1, \cdots, A_n)$, then an interpretation of $p$ is a predicate $\mathcal{I}(p) \subseteq \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$.

Then $s = (\mathcal{D}, \delta, \mathcal{I})$ is a *first-order state* (or *model*).

**Semantics of Equality**

**Definition (First-Order Equality)** The *equality* symbol $\doteq$ is declared as $\doteq (A, A)$ for any type $A \in \mathrm{TSym}$. The interpretation is fixed as $\mathcal{I}(\doteq) = \{ (d, d) : d \in \mathcal{D} \}$ ("referential equality", holds if arguments refer to identical objects).

### 5.2.1 Semantic Evaluation

Given a first-order state (model) $s$ and a variable assignment $\beta$, it is possible to evaluate first-order terms under $s$ and $\beta$.

**Definition (Valuation Function for First-Order Terms)** The valuation function $\mathrm{val}_{s,\beta} : \mathrm{Trm}_A \to \mathcal{D}^A$ for terms is defined inductively for each $A \in \mathrm{TSym}$:

$$\mathrm{val}_{s,\beta}(x) \coloneqq \beta(x)$$
$$\mathrm{val}_{s,\beta}\big(f(t_1, \cdots, t_r)\big) \coloneqq \mathcal{I}(f)\big(\mathrm{val}_{s,\beta}(t_1), \cdots, \mathrm{val}_{s,\beta}(t_r)\big)$$

Since $\mathrm{Trm} = \bigcup_{A \in \mathrm{TSym}} \mathrm{Trm}_a$, $\mathrm{val}_{s,\beta}$ is defined for all $t \in \mathrm{Trm}$.

**Definition (Valuation Function for First-Order Formulas)** The valuation function $\mathrm{val}_{s,\beta} : \mathrm{Fml} \to \{\mathfrak{t}, \mathfrak{f}\}$ for formulas is defined inductively:

$$\mathrm{val}_{s,\beta}\left(p(t_1, \cdots, t_r)\right) \coloneqq \begin{cases} \mathfrak{t} & \text{iff } \left(\mathrm{val}_{s,\beta}(t_1), \cdots, \mathrm{val}_{s,\beta}(t_r)\right) \in \mathcal{I}(p) \\ \mathfrak{f} & \text{otherwise} \end{cases}$$

$$\mathrm{val}_{s,\beta}(\neg\phi) \coloneqq \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{s,\beta}(\phi) = \mathfrak{f} \\ \mathfrak{f} & \text{otherwise} \end{cases}$$

$$\mathrm{val}_{s,\beta}(\phi \wedge \psi) \coloneqq \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{s,\beta}(\phi) = \mathfrak{t} \text{ and } \mathrm{val}_{s,\beta}(\psi) = \mathfrak{t} \\ \mathfrak{f} & \text{otherwise} \end{cases}$$

$$\mathrm{val}_{s,\beta}(\phi \vee \psi) \coloneqq \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{s,\beta}(\phi) = \mathfrak{t} \text{ or } \mathrm{val}_{s,\beta}(\psi) = \mathfrak{t} \\ \mathfrak{f} & \text{otherwise} \end{cases}$$

$$\mathrm{val}_{s,\beta}(\phi \to \psi) \coloneqq \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{s,\beta}(\phi) = \mathfrak{f} \text{ or } \mathrm{val}_{s,\beta}(\psi) = \mathfrak{t} \\ \mathfrak{f} & \text{otherwise} \end{cases}$$

$$\mathrm{val}_{s,\beta}(\forall x; \phi) \coloneqq \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{s,\beta^d_x}(\phi) = \mathfrak{t} \text{ for all } d \in \mathcal{D}^A, x : A \\ \mathfrak{f} & \text{otherwise} \end{cases}$$

$$\mathrm{val}_{s,\beta}(\forall x; \phi) \coloneqq \begin{cases} \mathfrak{t} & \text{iff } \mathrm{val}_{s,\beta^d_x}(\phi) = \mathfrak{t} \text{ for at least one } d \in \mathcal{D}^A, x : A \\ \mathfrak{f} & \text{otherwise} \end{cases}$$

### 5.2.2 Semantic Concepts

**Definition (Satisfiability, Truth, Validity)**

- A formula $\phi$ is *satisfiable* iff $\mathrm{val}_{s,\beta}(\phi)$ for at least one state/model.
- A formula $\phi$ is *true* (write $s \models \phi$) iff for all $\beta$ it holds that $\mathrm{val}_{s,\beta}(\phi) = \mathfrak{t}$.
- A formula $\phi$ is *valid* (write $\models \phi$) iff for all $s$ it holds that $s \models \phi$.

Notice: Closed formulas that are satisfiable are also true.

## 5.3 Sequent Calculus

Showing the validity of a FO formula by computation is impractical:

- There are uncountably many FO states.
- Even a single state may have an infinite domain,
- Even when when the domain is finite: recursion leads to an exponential number of cases.

So there is a need for a *syntactical* proof method using only symbols that occur already in the formula and avoiding recursive evaluation whenever possible.

The idea is to proof the validity of $\phi$ by *syntactic transformations* of $\phi$.

**Notation for Sequent** Sequent calculus is based on the notion of a *sequent*:

$$\underbrace{\psi_1, \cdots, \psi_m}_{\text{Antecedent}} \implies \underbrace{\phi_1, \cdots, \phi_n}_{\text{Succedent}}$$

which has the same meaning as

$$(\psi_1 \wedge \cdots \wedge \psi_m) \rightarrow (\phi_1 \vee \cdots \vee \phi_n)$$

The antecedent/succedent can be considered as (possibly empty) sets of formulas.

**Definition (Schema Variables, Schema Sequents)** Let $\phi_i, \psi_j, \cdots$ match formulas and $\Gamma, \Delta, \cdots$ match sets of formulas. Then infinitely many sequents can be characterized by the single *schematic sequent*

$$\Gamma \implies \phi \wedge \psi, \Gamma$$

that matches all sequents with a top-level occurrence of a conjunction in the succedent.
$\phi \wedge \psi$ is called the *main formula* and $\Gamma, \Delta$ *side formulas* of a sequent.

**Notation for Rules** *Rules* define a syntactic transformation schema for sequents that precisely reflects semantics of connectives. The general notation for such a rule with name *RuleName* is:

$$\text{RuleName}\ \frac{\overbrace{\Gamma_1 \implies \Delta_1 \quad \cdots \quad \Gamma_r \implies \Delta_r}^{\text{Premises}}}{\underbrace{\Gamma \implies \Delta}_{\text{Conclusion}}}$$

Meaning: To proof the conclusion, it is sufficient to proof all premises.

### 5.3.1 Overview: Calculus Rules

#### Propositional Calculus Rules

$$\text{notLeft}\ \frac{\Gamma \implies \phi, \Delta}{\Gamma, \neg\phi \implies \Delta} \qquad \text{notRight}\ \frac{\Gamma, \phi \implies \Delta}{\Gamma \implies \neg\phi, \Delta}$$

$$\text{andLeft}\ \frac{\Gamma, \phi, \psi \implies \Delta}{\Gamma, \phi \wedge \psi \implies \Delta} \qquad \text{andRight}\ \frac{\Gamma \implies \phi, \Delta \qquad \Gamma \implies \psi, \Delta}{\Gamma \implies \phi \wedge \psi, \Delta}$$

$$\text{orLeft}\ \frac{\Gamma, \phi \implies \Delta \qquad \Gamma, \psi \implies \Delta}{\Gamma, \phi \vee \psi \implies \Delta} \qquad \text{orRight}\ \frac{\Gamma \implies \phi, \psi, \Delta}{\Gamma \implies \phi \vee \psi, \Delta}$$

$$\text{impLeft}\ \frac{\Gamma \implies \phi, \Delta \qquad \Gamma, \psi \implies \Delta}{\Gamma, \phi \rightarrow \psi \implies \Delta} \qquad \text{impRight}\ \frac{\Gamma, \phi \implies \psi, \Delta}{\Gamma \implies \phi \rightarrow \psi, \Delta}$$

$$\text{close}\ \frac{}{\Gamma, \phi \implies \phi, \Delta} \qquad \text{true}\ \frac{}{\Gamma \implies \textbf{true}, \Delta} \qquad \text{false}\ \frac{}{\Gamma, \textbf{false} \implies \Delta}$$

**First-Order Calculus Rules**

$$\text{notLeft}\ \frac{\Gamma \implies \phi, \Delta}{\Gamma, \neg\phi \implies \Delta} \qquad \text{notRight}\ \frac{\Gamma, \phi \implies \Delta}{\Gamma \implies \neg\phi, \Delta}$$

$$\text{andLeft}\ \frac{\Gamma, \phi, \psi \implies \Delta}{\Gamma, \phi \wedge \psi \implies \Delta} \qquad \text{andRight}\ \frac{\Gamma \implies \phi, \Delta \qquad \Gamma \implies \psi, \Delta}{\Gamma \implies \phi \wedge \psi, \Delta}$$

$$\text{orLeft}\ \frac{\Gamma, \phi \implies \Delta \qquad \Gamma, \psi \implies \Delta}{\Gamma, \phi \vee \psi \implies \Delta} \qquad \text{orRight}\ \frac{\Gamma \implies \phi, \psi, \Delta}{\Gamma \implies \phi \vee \psi, \Delta}$$

$$\text{impLeft}\ \frac{\Gamma \implies \phi, \Delta \qquad \Gamma, \psi \implies \Delta}{\Gamma, \phi \to \psi \implies \Delta} \qquad \text{impRight}\ \frac{\Gamma, \phi \implies \psi, \Delta}{\Gamma \implies \phi \to \psi, \Delta}$$

$$\text{allLeft}\ \frac{\Gamma, \forall x; \phi, [x/t']\phi \implies \Delta}{\Gamma, \forall x; \phi \implies \Delta} \qquad \text{allRight}\ \frac{\Gamma \implies [x/c]\phi, \Delta}{\Gamma \implies \forall x; \phi, \Delta}$$

$$\text{exLeft}\ \frac{\Gamma, [x/c]\phi \implies \Delta}{\Gamma, \exists x; \phi \implies \Delta} \qquad \text{exRight}\ \frac{\Gamma \implies [x/t']\phi, \exists x; \phi, \Delta}{\Gamma \implies \exists x; \phi, \Delta}$$

$$\text{applyEqL}\ \frac{\Gamma, t \doteq t', [t/t']\phi \implies \Delta}{\Gamma, t \doteq t', \phi \implies \Delta} \qquad \text{applyEqR}\ \frac{\Gamma, t \doteq t' \implies [t/t']\phi, \Delta}{\Gamma, t \doteq t' \implies \phi, \Delta}$$

$$\text{close}\ \frac{}{\Gamma, \phi \implies \phi, \Delta} \qquad \text{true}\ \frac{}{\Gamma \implies \textbf{true}, \Delta} \qquad \text{false}\ \frac{}{\Gamma, \textbf{false} \implies \Delta}$$

$$\text{eqClose}\ \frac{}{\Gamma \implies t \doteq t, \Delta}$$

- $[t/t']\phi$ is the result of replacing every occurrence of $t$ in $\phi$ with $t'$.
- $t, t'$ are arbitrary variable-free terms of same type.
- $c$ is a new constant of same type as $x$ (that is not yet used on the current proof branch!).
- Equations can be reversed by symmetry.

### 5.3.2 Proofing

The goal to proof: $\mathcal{G} = \psi_1, \cdots, \psi_m \implies \phi_1, \cdots, \phi_n$.

- Find a rule $\mathcal{R}$ whose conclusion matches $\mathcal{G}$.
- Instantiate $\mathcal{R}$ such that the conclusion is identical to $\mathcal{G}$.
- Recursively find proofs for the resulting premises $\mathcal{G}_1, \cdots, \mathcal{G}_r$.
- The result is a tree structure with the goal as the root.
- Close a proof branch when a rule without a premise is encountered.

### Examples

#### Propositional

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \implies q, p}\,\text{close} \quad \dfrac{}{p, q \implies q}\,\text{close}}{p, p \to q \implies q}\,\text{impLeft}}{p \wedge (p \to q) \implies q}\,\text{andLeft}}{\implies \big(p \wedge (p \to q)\big) \to q}\,\text{impRight}$$

#### First-Order

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(c,d), \forall y; p(c,y) \implies p(c,d), \exists x; p(x,y)}\,\text{close}}{p(c,d), \forall y; p(c,y) \implies \exists x; p(x,d)}\,\text{exRight}}{\forall y; p(c,y) \implies \exists x; p(x,d)}\,\text{allLeft}}{\forall y; p(c,y) \implies \forall y; \exists x; p(x,y)}\,\text{allRight}}{\exists x; \forall y; p(x,y) \implies \forall y; \exists x; p(x,y)}\,\text{exLeft}$$

## 5.3.3 Soundness and Completeness

**Definition (Sound Sequent Rule)** A rule is *sound* (correct) iff the validity of its premises implies the validity of its conclusion.

**Definition (Soundness of Sequent Calculus)** A sequent calculus is *sound* iff a closed proof implies the validity of the formula corresponding to its root sequent.

**Definition (Completeness of Sequent Calculus)** A sequent calculus is *complete* iff for any valid formula $\phi$ a closed proof with root sequent $\implies \phi$ exists.

**Theorem** Soundness, CompletenessThe *sequent calculus* introduced above is sound and complete.

# 6 Java Modeling Language (JML)

This chapter covers deductive verification of Java source code:

1. Formally specifying Java programs and
2. proving Java programs are correct.

## 6.1 Running Examples

### 6.1.1 ATM

Listing 6.1: Running Example for JML 1

```
public class ATM {
    private BankCard insertedCard = null;
    private int wrongPinCounter = 0;
    private boolean customerAuthenticated = false;

    public void insertCard(BankCard card) { ... }
    public void enterPin(int pin) { ... }
    public int accountBalance() { ... }
    public int withdraw(int amount) { ... }
    public void ejectCard() { ... }
}

public class BankCard {
    private final int correctPin;
    private boolean invalid = false;
}
```

Listing 6.1 shows the code of the running “ATM”-Example.

### 6.1.2 Limited Integer Set

Listing 6.2: Running Example for JML 2

```
public class LimitedIntegerSet {
    public final int limit;
    private /*@ spec_public @*/ int a[];
    private /*@ spec_public @*/ int size = 0;

    public LimitedIntegerSet(int limit) {
        this.limit = limit;
        this.a = new int[this.limit];
    }

    public boolean add(int elem) { ... }
    public void remove(int elem) { ... }
    public /* pure @*/ boolean contains(int elem) { ... }
}
```

Listing 6.2 shows the code of the running “Limited Integer Set”-Example (with added specification modifiers that are explained later on).

## 6.2 Specification

- Specification target
  - System level specifications (requirement analysis, GUI, use cases) are important, but not subject of this document!
  - This focuses on *unit specification* – contracts between implementors on various levels:
    * Application level ↔ Application level
    * Application level ↔ Library level
    * Library level ↔ Library level
- Units in OOP settings are interfaces, classes and their methods.
- Focusing on methods, a specification must compromise the following aspects:
  - Result value
  - Initial values of formal parameters
  - Accessible part of pre- and post-state

### 6.2.1 Specifications as Contracts

A useful analogy to stress the different roles in a specification, view a specification as a *contract* between the method implementor and the user, between the caller and the callee: The callee guarantee a certain outcome provided that the caller guarantees some prerequisites.

### 6.2.2 Informal Specification (ATM Example, `enterPin`)

“Enter the PIN that belongs to the currently inserted bank card into the ATM. If a wrong PIN is entered three times in a row, the card is confiscated. After having entered the correct PIN, the customer is regarded as authenticated.”

### 6.2.3 Specification as Contract

The contract states what is guaranteed under which conditions.

**precondition** Card is inserted, user not yet authenticated, card is valid, PIN is correct.

**postcondition** User is authenticated.

**precondition** Card is inserted, user not yet authenticated, `wrongPinCounter < 2`, PIN is incorrect.

**postcondition** `wrongPinCounter` is increased by `1`, user is not authenticated.

**precondition** Card is inserted, user not yet authenticated, `wrongPinCounter >= 2`, PIN is incorrect.

**postcondition** Card is confiscated, card is made invalid, user is not authenticated.

**Definition (Satisfying Implementation)** A pre-/post-condition pair for a method $m$ is *satisfied by the implementation* of $m$ iff: When $m$ is called in any state that satisfies the precondition, then any terminating state of $m$ satisfies the postconditions.

Remarks:

- No guarantee for anything if the precondition is not satisfied.
- Termination may or may not be guaranteed.
- Terminating state may be reached by normal or abrupt termination (e.g. exception).

### 6.2.4 Formal Specification

Natural language specifications are very important and widely used, but this document focuses on formal specification where contracts are described with mathematical rigor.

- High degree of precision
  - formalization often exhibits omissions and inconsistencies
  - avoid ambiguities inherent to natural language
- Potential for automation of program analysis
  - run-time assertion checking
  - test case generated
  - program verification

## 6.3 Introduction to JML

JML (the *Java Modeling Language*) is a specification language that is tailored to Java and follows the philosophy to integrate specification and implementation in one single language. That is, JML is not external to Java, but an extension of the language, written in annotation-like comments.

JML annotations include:

- Preconditions (section 6.4.2)
- Postconditions (section 6.4.3 and 6.4.4)
- Class invariants (section 6.8.1)
- Additional modifiers (section 6.6)
- “Specification-only” fields (not covered)
- “Specification-only” methods (not covered)
- Loop invariants (section 6.10.1)

### 6.3.1 JML Integration

- JML annotations are attached directly to Java programs by writing them directly into the source code files.
- To ensure compatibility with the Java compiler, JML annotations are written inside special Java comments, thus are ignored by the standard Java compiler but recognized by JML tools.
- These special comments have to start with an  and everything following will be parsed by JML tools.

> **Warning:** There must not be a space between the start of the comment and the at! So only `/*@ ... */` and `//@ ...` will be recognized as JML annotations!

## 6.4 Specification Case

Specifications cases for a method must be directly in front of the method and have the form shown in listing 6.3 (using the ATM example). Multiple specification cases have to be connected using the keyword `also` where it is not necessary iff only one specification case is defined.

Listing 6.3: Specification Cases in JML

```
/*@
  @ <specification case one>
  @
  @ also
  @
  @ <specification case two>
  @
  @ ...
  @*/
public void enterPin(int pin) { ... }
```

A single specification case has the form shown in listing 6.4 (using the ATM example). The details of the specification case components are covered in the following sections.

Listing 6.4: Single Specification Case in JML

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires !insertedCard.invalid;
  @ requires pin == insertedCard.correctPin;
  @ ensures customerAuthenticated;
  @*/
public void enterPin(int pin) { ... }
```

### 6.4.1 Public Modifier

A `public` specification case:

- is accessible from all classes and interfaces and
- can only refer to `public` fields/methods of the class (to reference also private fields, they must be annotated with the JML modifier `spec_public` that is covered in section 6.6.1).

### 6.4.2 Preconditions

A specification case contains postcondition clauses of the form

```
requires b;
```

where b is a boolean JML expression. If multiple clauses are given, the precondition is a conjunction of all clauses. In the pre-state, the preconditions must be true to guarantee the postconditions (or exceptional behavior).

### 6.4.3 Normal Behavior

Each keyword that is ending in `behavior` opens a specification case. For the `normal_behavior` specification case, the called method guarantees to not throw an exception if the caller guarantees all preconditions of the specification case.

#### Postconditions

A normal specification case have multiple postcondition clauses of the form

```
ensures b;
```

where b is a boolean JML expression. If multiple clauses are given, the postcondition is a conjunction of all clauses.

If the preconditions of the specification case hold in the pre-state, the postconditions must hold in the post-state.

### 6.4.4 Exceptional Behavior

For an `exceptional_behavior` specification case, assuming the preconditions are fulfilled, it requires the method to throw an exception iff the pre-state satisfies the preconditions.

- The keyword `signals` specifies the post-state, depending on the type of the thrown exception.
- The keyword `signals_only` specified the type of the thrown exception.

JML specifications must fully separate normal/exceptional specification cases by suitable and disjoint preconditions!

#### Exception Types

An exceptional specification case can have at most one clause of the form

```
signals_only E1, ..., EN;
```

where E1 to EN are the exception types. The thrown exception must have one of these types.

#### Postconditions

An exceptional specification case can have several clauses of the form

```
signals (E)b;
```

where E is an exception type and b is a boolean JML expression. If an exception of type E is thrown, then b must hold.

### 6.4.5 Non-Termination

By default, both normal and exceptional specifications enforce termination. Non-termination can be allowed by the clause

```
diverges true;
```

in the specification. If the precondition of the specification case holds in the pre-state, then the method may or may not terminate.

## 6.5 Assignable

An assignable clause is a clause of the form

```
assignable loc1, ..., locN;
```

where `loc1` to `locN` are locations (e.g. class variables) that can be modified by the specified method. That is, no other values are allowed to change and must have the same value in the post-state as in the pre-state (but they may change temporarily during the execution of the method).

Some special values are `\nothing` and `\everything` where the first allows no modifications and the latter allows all modifications (default).

Specifying as less locations as possible makes validating the program more efficient and reduces the amount of specification in the postcondition.

## 6.6 Modifiers

JML extends the Java modifiers. Those modifiers have to be written between the “normal” modifiers, e.g. `private /*@ spec_public @*/int foo;`.

### 6.6.1 `spec_public`

- Public specifications are not able to access class fields that are not public.
- It is not desirable to make every fields public (encapsulation, etc.).
- The visibility of such fields can be increased with the modifier `spec_public` which keeps the Java visibility but, if necessary, rises the specification visibility to public.
- A different solution that is not covered here is to use specification-only fields.

### 6.6.2 `pure`

**Definition (Pure Method)** A Java method is called *pure* iff it has no side effects and always terminates. Specifically, it may create no new objects.

JML expressions are allowed pure methods. Those methods are annotated by the modifier `pure`, e.g. the `contains` method in the limited integer set example (see listing 6.2).

- `pure` puts obligation on the implementor to not cause side effects.

- It is possible to formally verify that a method is pure by writing a contract that expresses it and verifying it.
- `pure` implies `assignable \nothing;`.
- Assignable clauses are more fine-grained and can be local to specification cases where `pure` always effects the whole method.

### 6.6.3 `nullable`, `non_null`

Class fields, method parameters and returns types may be annotated with

- `nullable` – May or may not be `null`.
- `non_null` – Must not be `null` (default).

> **Warning:** The default `non_null` only applies in JML since 2008! Old JML tutorials/articles might use `nullable`-by-default semantics!

These modifiers are converted into class invariants by JML, see section 6.8.1.

## 6.7 JML Expressions

- Each side effect-free Java expression is a JML expression (every method call must be a pure method).
- If `E` is a side effect-free Java expression, then `\old(E)` is a JML expression referring to the value of `E` before method execution (useful in postconditions).
- If `a` and `b` are `boolean` JML expression, then all of the following are also `boolean` JML expressions:

  ```
  !a,   a && b,   a || b,   a ==> b,   a <==> b
  ```

  with the semantics negation, conjunction, disjunction, implication and equivalence.
- For a type `t` and `boolean` JML expressions `a` and `b`, all of the following are also `boolean` JML expressions:

  ```
  (\forall t x; b),   (\exists t x; b),   (\forall t x; a; b),   (\exists t x; a; b)
  ```

  with the semantics (in order):
  - For all `x` of type `t`, `b` is true.
  - There exists an `x` of type `t` such that `b` is true.
  - For all `x` of type `t` fulfilling `a`, `b` is true.
  - There exists an `x` of type `t` fulfilling `a` such that `b` is true.

  The `boolean a` is called *range predicate* and is syntactic sugar for standard FOL quantifiers.

**Notes:**

- Quantified JML expressions over nontrivial types also range over non-created objects which can be restricted with the expressions `\created(b1)` where `b1` is a bound variable.
- In postconditions, the return value (result value) of the method can be accessed via `\result`.

## 6.8 State Constraints

JML can also be used to specify constraints on the state of a class, e.g.:

- Consistency of redundant data representations (e.g. caching).
- Restrictions for efficiency (e.g. maintaining sortedness).

The constraints are global, i.e. all methods must preserve them.

### 6.8.1 Class Invariants

Such constraints can be formulated as *class constraints* that are written anywhere in the class in the following form:

```
public invariant b;
```

where b is a `boolean` JML expression. Technically, these constraints can be written anywhere in the class but it is common to write them in front of the affected properties. Multiply invariants per class are joined in a conjunction.

Invariant types:

- Instance invariants
  - Can refer to static fields.
  - Can refer to instance fields via quantifying over an explicit reference, e.g. `o.size`.
  - Can refer to instance fields of `this` object (qualified or unqualified).
  - Syntax: `instance invariant`
- Static invariants
  - Can refer to static fields.
  - Can refer to instance fields via quantifying over an explicit reference, e.g. `o.size`.
  - Cannot refer to instance fields of the object.
  - Syntax: `static invariant`

In classes, instance is default and static is default in interfaces.

**Semantics/Intuition** The class invariants are added as another pre- and postcondition to all methods of a class, so they do need to hold before and after the method call, but not necessarily during the execution.

## 6.9 Inheritance

All JML contracts (specification cases, class invariants) are inherited from superclasses to subclasses (a class must fulfill all contracts of its superclasses). Subclasses may also add specification cases to those of the superclasses, see listing 6.5.

Listing 6.5: Inheritance in JML

```
1 /*@ also
2   @ <specification case specific to subclass>
3   @*/
4 public void method() { ... }
```

## 6.10 Loops

This section only covers the syntax for proofing loop correctness in JML. For a more detailed dive into the formal backbone, lookup section 7.12.

### 6.10.1 Loop Invariants

Listing 6.6: Loop Invariants in JML

```
1 /*@ loop_invariant <expr>;
2   @ assignable <assign>;
3   @*/
```

A *loop invariant* is a statement that is true before and after every loop iteration. It is specified in JML as in listing 6.6 where the expression has to be a `boolean` JML expression and the assign statement is assumed (so it must be true, e.g. with `assignable \nothing;` it is possible to proof nonsense if its not true). KeY generates a proof obligation that ensures the correctness of the assignable clause.

### 6.10.2 Loop Termination

If a loop does not terminate, it is possible to proof its correctness as the initial, preserved and use case are immediately closable. This can lead to nonsense proofs.

Thus a way is needed to proof the termination of a loop based on an expression called *variant* that is getting smaller w.r.t. $\mathbb{N}$ is each iteration (it is not possible to decrease infinitely many often in the natural numbers). This decreasing gets added to the `loop_invariant` annotation with the following syntax:

```
decreasing v;
```

where v is the decreasing term and $v \geq 0$ is ensures by the loop body as well as v is strictly decreased by the loop body. This way, KeY can proof the termination of the loop.

# 7 Dynamic Logic (DL)

Note: Most of the following uses KeY pretty-printing format for heap access, see section 7.7.4.

- A logic and calculus that allows the expressions if properties like "if ..., then ... and afterwards ..." where some state can be modified by the formula itself is needed.
- One such *program logic* is *dynamic logic* (DL), combining first-order logic (FOL) with programs.
- In typed FOL, only static properties can be expressed, e.g. values of fields in a certain range. These properties consider only one program state at a time.
- The goal is to express the behavior of a program in terms of *state changes*, like in the property above. This yields the following requirements for such a logic (that are all satisfied by DL):
  - Can relate between different program states, i.e. before and after execution, within a single formula.
  - Program variables are represented by constant symbols that depend on the current program state.

## 7.1 Type Hierarchy

**Definition (Type Hierarchy)**

- Let $\mathrm{TSym}$ be a set of *type* symbols.
- Let $\sqsubseteq \subseteq \mathrm{TSym} \times \mathrm{TSym}$ be the *subtype relation* that is reflexive and transitive.
- It has an empty type $\bot$ and a universal type $\top$ with

  $$\bot \sqsubseteq A \sqsubseteq \top$$

  for all $A \in \mathrm{TSym}$. Terms of type $\bot$ are not allowed!

Then $\mathcal{T} = (\mathrm{TSym}, \sqsubseteq)$ is a *type hierarchy*.

The Java type hierarchy is shown in figure 7.1 (not all primitive types listed) where each interface and class in the API and in the target program becomes a type with an appropriate subtype relation.

## 7.2 Syntax

**Definition (Signature of Dynamic Logic)** The signature is given as $\Sigma = (\mathrm{PSym}, \mathrm{FSym}, \mathrm{VSym}, \mathrm{ProgVSym})$ with $\mathrm{FSym} \cap \mathrm{ProgVSym} = \emptyset$.

- $\mathrm{PSym}$ Predicate symbols (rigid), e.g. $\mathrm{PSym} = \{$ `>`, `>=`, $\cdots\}$
- $\mathrm{FSym}$ Function symbols (rigid), e.g. $\mathrm{FSym} = \{$ `+`, `-`, `*`, `0`, `1`, $\cdots\}$
- $\mathrm{ProgVSym}$ Program variables (flexible), e.g. $\mathrm{ProgVSym} = \{$ `i`, `j`, `k`, $\cdots\}$

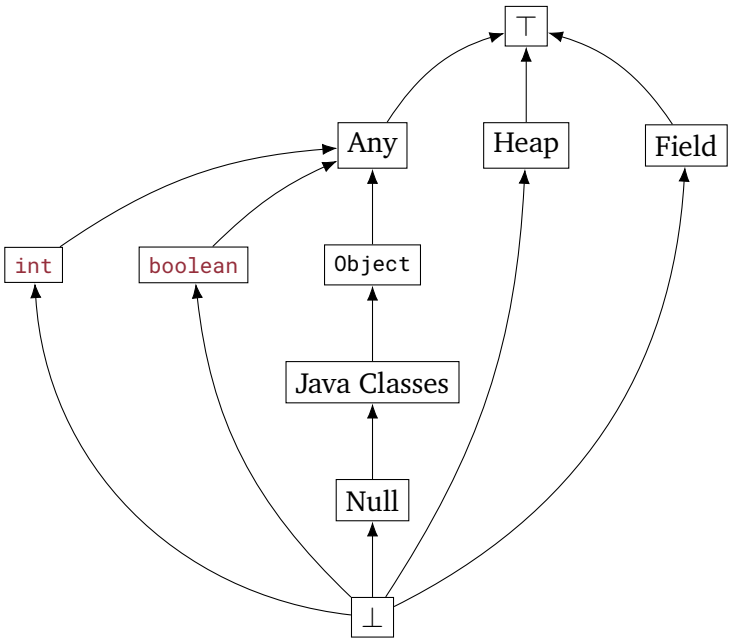


Figure 7.1: Java Type Hierarchy

### 7.2.1 Variables

Program variables cannot be first-order variables and are state-dependent constant symbols where the value is changeable by a program (state-dependent = non-rigid = flexible).

- The signature of dynamic logic is defined like in FOL, but with additional program variables for each type.
- Rigid signature symbols have the same interpretation in all program states (first-order variables and builtin functions and predicates).
  - These occur in quantifiers and similar.
  - Must not occur in programs!
- Non-rigid (flexible) symbols depend on the state, i.e. the interpretation differs from state to state (program variables).
  - Cannot be quantified.
  - May occur in programs and formulas.
  - Can be declared in the code inside a DL formula or outside in the KeY file.
- Any term that contains at least one flexible symbol is also flexible.

### 7.2.2 Terms

**Definition (Dynamic Logic Terms)** The typed first-order terms are defined as in FOL. In addition, a program variable v of type $A$ is a term in $\mathrm{Trm}_A$.

### 7.2.3 Modalities

Dynamic logic extends FOL with two additional operations:

- $\langle \mathsf{p} \rangle \phi$ (diamond)
- $[\mathsf{p}] \phi$ (box)

with a program $\mathsf{p}$ and another DL formula $\phi$.
Intuitive meaning:

- $\langle \mathsf{p} \rangle \phi$: $\mathsf{p}$ terminates and formula $\phi$ holds in the final state (total correctness).
- $[\mathsf{p}] \phi$: If $\mathsf{p}$ terminates, then formula $\phi$ holds in the final state (partial correctness).

As Java programs are deterministic, if a Java program terminates, then exactly one state is reached from a given initial state.

### 7.2.4 Formulas

**Definition (Dynamic Logic Formulas)** Dynamic logic formulas are given by the following inductive definition:

- Each FOL formula is also a DL formula.
- If $\mathsf{p}$ is a program and $\phi$ is a DL formula, then all of the following are also DL formulas:

$$\langle \mathsf{p} \rangle \phi, \quad [\mathsf{p}] \phi$$

- DL formulas are closed under FOL quantifiers and connectives.

## 7.3 Semantics

- A first-order state (model) is considered as a program state.
- In dynamic logic, the interpretation of flexible program variable differs from one program state to another.
- The interpretation of rigid symbols is the same in all states.

**Definition (Program State)** A *program state* $s$ is a specific first-order state $(\mathcal{D}, \delta, \mathcal{I})$.
For a fixed $\mathcal{D}$, $\delta$ the set $\mathcal{S}$ denotes the set of all program states such that any two states $s_1, s_2 \in \mathcal{S}$:

- Coincide on $\mathcal{D}, \delta$ and
- may only differ in the interpretation $\mathcal{I}$ of flexible program variables. Thus, $\mathcal{I}$ only records the values of $\mathsf{v} \in \mathrm{ProgVSym}$.

### 7.3.1 Kripke Structure

**Definition (First-Order Kripke Structure)** A *Kripke structure* or *labeled transition system* $\mathcal{K} = (\mathcal{S}, \rho)$ consists of:

- States (first-order models) $s = (\mathcal{D}, \delta, \mathcal{I}) \in \mathcal{S}$ and
- Transition relation $\rho : \text{Program} \to (\mathcal{S} \rightharpoonup \mathcal{S})$ with $\rho(\mathsf{p})(s_1) = s_2$ iff the program $\mathsf{p}$, executed in state $s_1$, terminates normally and its final state is $s_2$. Otherwise, the transition is undefined.
- $\rho$ is the formal semantics of programs $\mathsf{p} \in \text{Program}$.
- $\rho(\mathsf{p})(s)$ may be undefined (denoted by $\rightharpoonup$), $\mathsf{p}$ might not terminate when started in some state $s$.
- Java programs are deterministic (unlike PROMELA), thus $\rho(\mathsf{p})$ has at most one value.
- First-order Kripke structures are infinite (one transition for each program with infinitely many programs).
- The states only differ in the interpretation of the program variables (rigid symbols have the same interpretation in all states).

### 7.3.2 Semantic Evaluation

#### Formulas

**Definition (Validity Relation of DL Formulas)** Let $\mathcal{K}$ be a Kripke structure, $s \in \mathcal{S}$ a state, $\beta$ a variable assignment, $\mathsf{p} \in \text{Program}$ a program, $\phi$ and DL formula and $\psi$ and FOL formula. Then the validity relation is defined as:

$$\begin{array}{lll} (\mathcal{K}, s, \beta) \models \langle \mathsf{p} \rangle \phi & \iff & p(\mathsf{p})(s) \text{ is defined and } \big(\mathcal{K}, \rho(\mathsf{p})(s), \beta\big) \models \phi \\ (\mathcal{K}, s, \beta) \models [\mathsf{p}] \phi & \iff & \big(\mathcal{K}, \rho(\mathsf{p})(s), \beta\big) \models \phi \text{ when } p(\mathsf{p})(s) \text{ is defined} \\ (\mathcal{K}, s, \beta) \models \psi & \iff & \text{val}_{s,\beta}(\psi) = \mathfrak{t} \end{array}$$

- Duality: $\langle \mathsf{p} \rangle \phi \iff \neg [\mathsf{p}] \neg \phi$
- Implication: $\langle \mathsf{p} \rangle \phi \implies [\mathsf{p}] \phi$
  Total correctness implies partial correctness, holds only for deterministic programs.

#### Programs

In a Kripke structure $\mathcal{K} = (\mathcal{S}, \rho)$, $\rho$ defines the semantics of programs $\mathsf{p} \in \text{Program}$. It is a verify tedious task to define $\rho$ for Java and thus it is not done here.

### 7.3.3 Sequents

**Definition (DL Sequent)** Let $\Gamma = \{ \phi_1, \cdots, \phi_n \}$ and $\Delta = \{ \psi_1, \cdots, \psi_m \}$ be sets of DL formulas where all first-order variables are bound and let $s \in \mathcal{S}$ be a program state. Then

$$(\mathcal{K}, s) \models (\Gamma \implies \Delta) \quad \text{iff} \quad (\mathcal{K}, ) \models (\phi_1 \wedge \cdots \wedge \phi_n) \to (\psi_1 \vee \cdots \vee \psi_m)$$

like in propositional logic and FOL sequents.

**Definition (Validity of DL Sequents)** A sequent $\Gamma \implies \Delta$ over DL formulas is valid iff

$$(\mathcal{K}, s) \models (\Gamma \implies \Delta)$$

in all states $s \in \mathcal{S}$ and Kripke structures $\mathcal{K}$.

Note that the initial value of program variables can be any domain value!

## 7.4 KeY Input File

## 7.5 Symbolic Execution

- In DL symbolic execution, the natural control flow of a program is followed.
- Values of some variables unknown: represent symbolic state.
- The general form of rule conclusion in symbolic calculus is

  $$\langle \texttt{stmt; rest} \rangle \phi, \quad [\texttt{stmt; rest}] \phi$$

  where the rules symbolically execute the first statement `stmt` (called *active statement*).
- Repeated application of such rules corresponds to *symbolic program execution*.

### 7.5.1 Rules

**Conditional**

$$\text{if}\ \frac{\Gamma, \texttt{b} \doteq \texttt{true} \implies \langle \texttt{p; rest} \rangle \phi, \Delta \qquad \Gamma, \texttt{b} \doteq \texttt{false} \implies \langle \texttt{q; rest} \rangle \phi, \Delta}{\Gamma \implies \langle \texttt{if (b) \{ p \} else \{ q \}; rest} \rangle \phi, \Delta}\ \texttt{b}\ \text{simple}$$

**Loops: Unwind**

$$\text{unwindLoop}\ \frac{\Gamma \implies \langle \texttt{if (b) \{ p; while (b) \{ p \} \}; rest} \rangle \phi, \Delta}{\Gamma \implies \langle \texttt{while (b) \{ p \}; rest} \rangle \phi, \Delta}$$

**Assignment**

$$\text{assign}\ \frac{\Gamma \implies \{ \texttt{x} := \texttt{t} \} \langle \texttt{rest} \rangle \phi, \Delta}{\Gamma \implies \langle \texttt{x = t; rest} \rangle \phi, \Delta}\ \texttt{t}\ \text{simple}$$

**Field Assignment**

$$\text{assign}\ \frac{\Gamma \implies \{ \text{heap} := \text{store}(\text{heap}, \texttt{o}, \texttt{f}, t) \} \langle \texttt{rest} \rangle \phi, \Delta}{\Gamma \implies \langle \texttt{o.f = t; rest} \rangle \phi, \Delta}\ \texttt{t}\ \text{simple}$$

**Local Code Transformation**

$$\text{evalOrderIteratedAssgnmt}\ \frac{\Gamma \implies \langle \texttt{y = t; x = y;} \ \omega \rangle \phi, \Delta}{\Gamma \implies \langle \texttt{x = y = t;} \ \omega \rangle \phi, \Delta}\ \texttt{t}\ \text{simple}$$

#### Complex Conditional

$$\text{ifEval}\,\frac{\Gamma \Longrightarrow \langle \texttt{boolean v0; v0 = b; if (v0) p; } \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{if (b) p; } \omega\rangle\phi, \Delta}\,\texttt{b complex}$$

#### Try-Catch

$$\text{tryThrow}\,\frac{\Gamma \Longrightarrow \langle \pi\ \texttt{if (e instanceof T) \{ try \{ x = e; q \} finally \{ r \} \} else \{ r; throw e; \} } \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{try \{ throw e; p \} catch (T x) \{ q \} finally \{ r \} } \omega\rangle\phi, \Delta}$$

#### Method Call: Inlining, Void

$$\text{methodBodyExpand}\,\frac{\Gamma \Longrightarrow \langle \pi\ \texttt{method-frame(source=C, this=C):\{ body \} } \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{c.m(p\#1, ..., p\#N)@C; } \omega\rangle\phi, \Delta}$$

#### Method Call: Contract, Void

$$\text{methodContract}\,\frac{\begin{array}{c}\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\texttt{preExc}), \Delta \\ \Gamma \Longrightarrow \mathcal{U}\nu_{\texttt{normal}}\big(\mathcal{F}(\texttt{postNormal}) \rightarrow \langle \pi\ \texttt{p } \omega\rangle\phi\big), \Delta \\ \Gamma \Longrightarrow \mathcal{U}\nu_{\texttt{exc}}\Big(\big(\texttt{exc} \doteq \texttt{null} \wedge \mathcal{F}(\texttt{postExc})\big) \rightarrow \langle \pi\ \texttt{throw exc; p } \omega\rangle\phi\Big), \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \texttt{result = m(arg1, ..., argN); p } \omega\rangle\phi, \Delta}$$

## 7.6 Updates

**Definition (Syntax of Updates, Updated Terms/Formulas)** Let $\mathsf{v}$ be a program variable, $t$ a FOL term that is type-conformant to $\mathsf{v}$, $t'$ any FOL term and $\phi$ any DL formula, then:

- $\{\, \mathsf{v} := t \,\}$ is an update (of $\mathsf{v}$ to $t$),
- $\{\, \mathsf{v} := t \,\}t'$ a DL term and
- $\{\, \mathsf{v} := t \,\}\phi$ a DL formula.

**Definition (Semantics of Updates)** Let a state $s$ interpret the program variable $\mathsf{v}$ with $\mathcal{I}_s(\mathsf{v})$ and let $\beta$ be a variable assignment for free first-order variables in $t$. Then the semantics $\rho$ of an update is defined as:

$$\rho\big(\{\, \mathsf{v} := t \,\}\big)(s) = s'$$

where $s'$ is identical to $s$ except $\mathcal{I}_{s'}(\mathsf{v}) = \mathrm{val}_{s,\beta}(t)$.

- Update semantics are almost identical to those of the assignment statement.
- The value of an update depends on $s$ and on first-order variables in $t$.
- Updates are not assignments! Assignments may have side-effects, updates do not.
- Updates are not equations, they change the value of $\mathsf{v}$.

### 7.6.1 Effects

The computations of an update on a program $\mathsf{p}$ are delayed until it is symbolically executed. The rewrite rules for an update followed by…

- program variable:

$$\begin{cases} \{\,\mathsf{x} \coloneqq t\,\}y \rightsquigarrow \mathsf{y} & \mathsf{x} \neq \mathsf{y} \\ \{\,\mathsf{x} \coloneqq t\,\}x \rightsquigarrow t & \text{update happens} \end{cases}$$

- logical variable:

$$\{\,\mathsf{x} \coloneqq t\,\}w \rightsquigarrow w$$

- complex term ($f \in \mathrm{FSym}$):

$$\{\,\mathsf{x} \coloneqq t\,\}f(t_1, \cdots, t_n) \rightsquigarrow f\big(\{\,\mathsf{x} \coloneqq t\,\}t_1, \cdots, \{\,\mathsf{x} \coloneqq t\,\}t_n\big)$$

- FOL formula:

$$\begin{cases} \{\,\mathsf{x} \coloneqq t\,\}(\phi \,\&\, \psi) \rightsquigarrow \{\,\mathsf{x} \coloneqq t\,\}\phi \,\&\, \{\,\mathsf{x} \coloneqq t\,\}\psi \\ \quad \cdots \\ \{\,\mathsf{x} \coloneqq t\,\}(\forall y; \phi) \rightsquigarrow \forall y; \big(\{\,\mathsf{x} \coloneqq t\,\}\phi\big) \end{cases}$$

- program formula: no rewrite needed!

### 7.6.2 Assignment Rule

$$\text{assign}\,\frac{\Gamma \Longrightarrow \{\,\mathsf{x} \coloneqq \mathtt{t}\,\}\langle \mathtt{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathtt{x = t; \ rest}\rangle\phi, \Delta}$$

- Simple, not variable renaming or similar.
- Works as long as $t$ does not have any side effects.
- Special case is needed for calculations in the assignment etc.

### 7.6.3 Parallel Updates

If a variable is assigned multiple times, those several sequential updates have to be composed into a parallel update.

**Definition (Parallel Update)** A *parallel update* is an expression of the form $\{\,\mathsf{v}_1 \coloneqq r_1 \parallel \cdots \parallel \mathsf{v}_n \coloneqq r_n\,\}$ where each $\{\,\mathsf{v}_i \coloneqq r_i\,\}$ is a simple update.

- All $r_i$ are computed in the old state before the update is applied.
- All updates of the program variables $\mathsf{v}_i$ are executed simultaneously.
- Upon conflict $\mathsf{v}_i = \mathsf{v}_j$, $r_i \neq r_j$, the later update ($\max\{\,i, j\,\}$) wins.

### 7.6.4 Quantifier

Quantifying over a program variable is not possible, but it is possible to quantify over a FOL variable and then update the program variable accordingly:

$$\forall i_0; \{ \mathtt{i} \coloneqq i_0 \} \langle \cdots \mathtt{i} \cdots \rangle \phi$$

### 7.6.5 Anonymising Updates

Given some assignable clause

```
assignable modified;
```

the part of the prestate $\mathcal{U}$ that is unmodified by the execution should be kept. The assignable clause tells what could possibly be modified.

An *anonymising update* $\nu$ eases all information about the modified locations

$$\nu_{\text{modified}} = \{ \mathtt{l}_1 \coloneqq \mathtt{c}_1 \parallel \cdots \parallel \mathtt{l}_n \coloneqq \mathtt{c}_n \}$$

and sets them to a new constant symbol $\mathtt{c}_i$ for each location $\mathtt{l}_i$ in the assignable clause (`modified`).

**Anonymising Heap Locations** It must also be possible to erase some heap values without throwing everything away, e.g. for the assignable clause

```
assignable o.a, this.*;
```

**Definition (Anonymising Function)** Let anon : Heap × LocSet × Heap → Heap be a function that returns the heap that coincided with the first heap argument on all locations except those specified in the location set. The latter attain their value specified by the second heap argument.

Let *ls* of type LocSet be a set of location key ⊆ Object × Field:

$$\text{select}(\text{anon}(h_1, ls, h_2), o, f) = \begin{cases} \text{select}(h_2, o, f) & \text{iff } \big((o,f) \in ls \wedge f \neq \texttt{<created>}\big) \vee (o,f) \in \mathit{freshLocs}(h_1) \\ \text{select}(h_1, o, f) & \text{otherwise} \end{cases}$$

## 7.7 Modeling the Heap

In Java, values of reference types live on the heap.

- Can change dynamic during symbolic program execution.
- In each program state, the heap associated objects, fields, values.

### 7.7.1 Modeling Fields

- For every Java reference type $C$, there is a FOL type $C$, e.g. `Person`.
- For each field $f$ there is a unique constant $f \in \mathrm{FSym}$ of type Field, e.g. `age`.
- The type domain of all possible objects of a reference type $C$ is $\mathcal{D}^C$, e.g. $\mathcal{D}^{\texttt{Person}}$.
- A heap relates objects and fields to values.

**Reading Fields** Signature in DL: `Any select(Heap, Object, Field)`

- The return type is `Any`, so a cast to the appropriate type is needed.
- For casting, use `type :: select(Heap, Object, Field)`
- It is possible that a field with the same name exists in different classes. Thus, prepend the class name to the field name, e.g. `Person::$age`.

Example:

**Java** `p.age >= 0`

**FOL** `int ::select(heap, p, Person::$age) >= 0`

**KeY** `p.age >= 0` (using pretty-printing)

**Writing to Fields** Signature in DL: `Heap store(Heap, Object, Field, Any)`

Example:

**Java** `p.age = 42`

**FOL** `store(heap, p, Person::$age, 42)`

**KeY** `p.age := 42` (using pretty-printing)

### 7.7.2 Field Updates

DL has a reserved flexible constant Heapheap that stores the Java heap under verification for read/write field access.

$$\text{assign}\ \frac{\Gamma \implies \{\,\text{heap} := \text{store}(\text{heap}, \mathtt{o}, \mathtt{f}, t)\,\}\langle \mathtt{rest}\rangle\phi, \Delta}{\Gamma \implies \langle \mathtt{o.f = t; \ rest}\rangle\phi, \Delta}$$

### 7.7.3 Field Access

To compute the value of a field $f$ of an object $o$ of type $A$ in a given heap, perform a lookup in the heap using the pair $(o, f)$ as key/index.

### 7.7.4 Pretty-Printing

**Reading** `A :: select(heap, o, f)` is shown as `o.f`

**Modifying** `select(store(heap, o, f, 17), u, f)` is shown as `u.f@heap[o.f := 17]`

## 7.8 Aliasing

Naive reference aliasing causes a proof split (on $o \doteq u$) at each access

$$\implies \mathtt{o.age} \doteq 1 \mathtt{\ ->\ } \langle \mathtt{u.age = 2;}\rangle \mathtt{o.age} \doteq \mathtt{u.age}$$

To avoid case distinction on $o \doteq u$ in the proof, delayed state computation until until it is clear what is required is required as well as simplification of heap terms.

### 7.8.1 Static Fields

**No Compile-Time Constant** For each class $C$ with a static field $f$ of type $A$ that is not a compile-time constant, there is:

- A unique constant $C.f$ of type Field,
- whose value `v` is stored on the heap as `store(heap, null, C.f, v)`
- where $C$ is the fully qualified class name.

So static values are also stored on the heap!

#### Compile-Time Constant

For each class $C$ with a final static field $f$ of type $A$ that is a compile-time constant, there is

- a constant $C.f$ of type $A$ whose interpretation $\mathcal{I}_s(C.f)$ is the same for all states s.

So static compile-time constants are not stored on the heap!

### 7.8.2 `this` and `self`

Reserved name for the object whose Java code is currently executed:

**Java** `Object this;`

**JML** `Object this;`

**KeY** Arbitrary, but by convention often `Object self;`. The name is arbitrary as long as it is unique: only a flexible constant.

The default assumption in JML-KeY translation is `self != null`.

### 7.8.3 Arrays

- The Java type hierarchy includes array types that occur in the current program (only for finiteness).
- Types are ordered according to Java subtyping rules.
- Value of an entry in an array `T[] a;` defined in class $C$ depends on the reference `a` to the array in $C$ and the index.
- A flexible function `arr` : `int` $\rightarrow$ Field is an injective mapping from indices to fields (an array entry is modeled as a field).
- Array elements are stored on the heap, e.g. the value of `a[i]` on the heap `store(heap, a, arr(i), 17)` is 17.
- Arrays `a` and `b` can refer to the same object (aliases).
- KeY implements simplification and evaluation rules for array locations.

## 7.9 Abrupt Termination

Abrupt termination does not count as a normal termination, so a distinction between normal and exceptional termination is needed. This can be done by extending the definition of the two DL modalities:

- $\langle \mathsf{p} \rangle \phi$: p terminated *normally* and formula $\phi$ holds in the final state (total correctness).
- $[\mathsf{p}] \phi$: If p terminates *normally*, then formula $\phi$ holds in the final state (partial correctness).

### 7.9.1 Null Pointers

There are no "exceptions" in FOL, so a possibility that $o \doteq \texttt{null}$ has to be modeled in the term `o.f`:

- KeY branches over `o != null` on each field acces.
- The value of the term `null.f` is defined, but unknown.
- It is possible to proof `null.f` $\doteq$ `null.f`.

### 7.9.2 Try-Catch

$$\text{tryThrow}\ \frac{\Gamma \Longrightarrow \langle \pi\ \texttt{if (e instanceof T) \{ try \{ x = e; q \} finally \{ r \} \} else \{ r; throw e; \}}\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{try \{ throw e; p \} catch (T x) \{ q \} finally \{ r \}}\ \omega \rangle \phi, \Delta}$$

## 7.10 Complex Expressions

- Java expressions may contain assignment operators with a side effects.
- Java expressions can be complex, nested, contain method calls, etc.
- FOL terms have not side effect on the state.
- Idea: Simplify complex expression with side effects to side effect-free, simple expressions before translation to FOL.

Local code transformation:

$$\text{evalOrderIteratedAssgnmt}\ \frac{\Gamma \Longrightarrow \langle \texttt{y = t; x = y;}\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{x = y = t;}\ \omega \rangle \phi, \Delta}\ \mathsf{t}\ \text{simple}$$

Temporary variables tore the result of evaluated subexpressions:

$$\text{ifEval}\ \frac{\Gamma \Longrightarrow \langle \texttt{boolean v0; v0 = b; if (v0) p;}\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{if (b) p;}\ \omega \rangle \phi, \Delta}\ \mathsf{b}\ \text{complex}$$

Guards if conditionals/loops are always evaluated before condition/unwind rules are applied.

## 7.11 Method Calls

The syntax of a method call with actual (symbol) parameters `arg1`, $\cdots$ , `argN` is:

$$\{\,\mathtt{c} \coloneqq t\,\}\langle \mathtt{c.m(arg1,\ ...,\ argN)}\rangle\phi$$

where `m` is declared as `void m(A1 p1, ..., AN pN)`.

Action of the DL rule *methodCall* (type conformance of `argI` to `AI` is guaranteed by the Java compiler):

1. For each formal parameter `pI` of `m`: Declare and initialize a new local variable `AI p#i = argI;`
2. Lookup the implementation class $C$ of `m` and split the proof when more than one compile-time implementation is possible (necessitated by dynamic dispatch in general).
3. Create a statically resolved method invocation `c.m(p#1, ..., p#N)@C`.

The proof splits are required as Java has complex rules for localization of fields and method implementations (polymorphism, late binding, scoping, visibility, ...).

### 7.11.1 Method Execution by Inlining (void Methods)

1. Execute code binding actual to formal parameters `AI p#i = argI;`
2. Call DL rule *methodBodyExpand*
   a) Rename each occurrence of `pI` in body into `p#i`.
   b) Replace method invocation by the method frame and method body.

$$\text{methodBodyExpand}\,\frac{\Gamma \Longrightarrow \langle\pi\ \mathtt{method\text{-}frame(source=C,\ this=C):\{\ body\ \}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\pi\ \mathtt{c.m(p\#1,\ ...,\ p\#N)@C;}\ \omega\rangle\phi, \Delta}$$

#### Limitations of Inlining

- Source code might not be available (e.g. for commercial APIs or OS-specific methods).
- Method is invoked multiple times in a program:
  - $n$ nested methods with 2 calls each result in $2^n$ inlined method bodies.
  - Avoid multiple symbolic execution of identical code!
- Cannot handle unbounded recursion.

A solution to this is to use contracts instead of the method implementation. Steps:

1. Show that the requires clause is satisfied.
2. Obtain the postcondition from the ensures clause.
3. Delete updates to assignable locations from symbolic state.

This subject is continued in section 7.11.3.

### 7.11.2 Object Initialization

Assume that the domain $\mathcal{D}$ is the same in all states of $\mathcal{K} = (\mathcal{S}, \rho)$, i.e. object creation with `new` does not change the domain! One consequence is that the validity of rigid FOL formulas is unaffected by programs containing `new`.

Realizing constant domain assumption:

- Implicitly declare a flexible function `boolean <created>(Object);` that returns `true` iff the argument object has been created.
- Initialized as $\mathcal{I}(\texttt{<created>})(o) = \mathfrak{f}$ for all $o \in \mathcal{D}$.
- Object creation is modeled as an update $\{\, \texttt{heap} := \texttt{create(heap), o} \,\}$ for the next "free" `o`. This essentially sets `<created>` to `true`.

Java has complex rules for object initialization, e.g. the chain of constructors calls up until Object, implicit `super` call, visibility issues, initialization sequence, etc. These initialization rules are coded in methods `<createObject>()`, `<init>()`, ... which themselves are symbolically executed.

### 7.11.3 Method Contract Rule

See section 7.11.1 for the motivation for using method contracts.

Listing 7.1: Method Contract Rule Specification Example

```
1  /*@ public normal_behavior
2    @ requires   preNormal;
3    @ ensures    postNormal;
4    @ assignable modified;
5    @
6    @ also
7    @
8    @ public exceptional_behavior
9    @ requires   preExc;
10   @ signals    (Exception exc) postExc;
11   @ assignable modified;
12   @*
```

Given the generic specification in listing 7.1, the method contract rule for the normal behavior case is given as

$$\text{methodContractNormal}\ \frac{\begin{array}{c}\Gamma \implies \mathcal{U}\mathcal{F}(\texttt{preNormal}), \Delta \\ \Gamma \implies \mathcal{U}\nu_{\texttt{modified}}\big(\mathcal{F}(\texttt{postNormal}) \rightarrow \langle \pi\ \texttt{p}\ \omega \rangle \phi\big), \Delta\end{array}}{\Gamma \implies \mathcal{U}\langle \pi\ \texttt{result = m(arg1, ..., argN); p}\ \omega \rangle \phi, \Delta}$$

where $\mathcal{F}(\cdot)$ denotes the translation from JML to DL and $\nu_{\texttt{modified}}$ is the anonymising update (see section 7.6.5).

The method contract rule for the exceptional behavior case is given as:

$$\text{methodContractExc}\ \frac{\begin{array}{c}\Gamma \implies \mathcal{U}\mathcal{F}(\texttt{preExc}), \Delta \\ \Gamma \implies \mathcal{U}\nu_{\texttt{modified}}\Big(\big(\texttt{exc} \neq \texttt{null} \wedge \mathcal{F}(\texttt{postExc})\big) \rightarrow \langle \pi\ \texttt{throw exc; p}\ \omega \rangle \phi\Big), \Delta\end{array}}{\Gamma \implies \mathcal{U}\langle \pi\ \texttt{result = m(arg1, ..., argN); p}\ \omega \rangle \phi, \Delta}$$

Combined they yield the rule *methodContract*:

$$\text{methodContract}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\texttt{preExc}), \Delta \\ \Gamma \Longrightarrow \mathcal{U}\nu_{\texttt{normal}}\big(\mathcal{F}(\texttt{postNormal}) \rightarrow \langle \pi\ \texttt{p}\ \omega\rangle\phi\big), \Delta \\ \Gamma \Longrightarrow \mathcal{U}\nu_{\texttt{exc}}\Big(\big(\texttt{exc} \neq \texttt{null} \wedge \mathcal{F}(\texttt{postExc})\big) \rightarrow \langle \pi\ \texttt{throw exc; p}\ \omega\rangle\phi\Big), \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \texttt{result = m(arg1, ..., argN); p}\ \omega\rangle\phi, \Delta}$$

### 7.11.4 Proofing Correctness of Method Contracts

The usage of the contract rule depends on the correctness of the contract. The correctness of a (normal behavior) JML contract can be shown by proofing

$$\mathcal{F}(\texttt{preNormal}) \rightarrow \Big\langle \texttt{try \{ o.m(); \} catch(Exception e) \{ exc = e; \}} \Big\rangle \big(\texttt{exc} \neq \texttt{null} \wedge \mathcal{F}(\texttt{postNormal}) \wedge \textit{assignableCorrect}\big)$$

## 7.12 Loop Invariants

### 7.12.1 Deriving Loop Invariants

### 7.12.2 Context Loss

## 7.13 Understanding Unclosed Proofs

Reasons why a proof may not close:

- Buggy or incomplete specification
- Bug in program
- Maximal number of steps reached; restart or increase number of steps
- Automatic proof search fails; manual rule application necessary

Understanding open proof goals:

- Follow the control flow from the proof root to the open goal
- Branch labels give useful hints
- Identify unprovable part of post condition or invariant
- Sequent remains always in “pre-state”
  Constraints on program variables refer to the value at the start of a program (except the formula is behind update or modality).
- NB: $\Gamma \Longrightarrow \texttt{o} \doteq \texttt{null}, \Delta$ is equivalent to $\Gamma, \texttt{o} \neq \texttt{null} \Longrightarrow \Delta$.

### 7.13.1 Model Search