

# Generativní programování pro vestavěné systémy

Marek Paška

# Přehled

1. motivace a cíle
2. nástroje, které používám
3. případová studie

# Software ve vestavěných zař.

- omezené zdroje (tlak na cenu zařízení)
- spolehlivý
  - selhání může mít vážné následky
  - chyba se těžko opravuje
- reaguje na podněty z okolí
  - reaktivní
  - reálný čas

# Přístup k vývoji

- desktopový / serverový software:
  - čas programátora vždy dražší než HW zdroje
  - nové a nové vrstvy abstrakce (VM, skriptovací jazyky)
- vestavěný software:
  - roli hrají výrobní náklady
  - extrémně konzervativní (assembler, čisté C)
  - Ada?
  - Java? (režie, JIT, RT)

# Zbraně vývojáře

- generativní programování
  - od časů Fortranu používáme všichni
- formální metody
  - mocné ale „těžké“ - speciální nástroje
  - výsledný kód nutně nemusí mít vlastnosti ukázané na formálním modelu
- pokročilé praktiky softwarového inženýrství
  - aspektově orientované programování
  - design by contract

# Idea: zlepšit vývojový proces

- popsat zamýšlený program v nějakém „vhodném“ jazyce
  - vhodná úroveň abstrakce
  - dostatečná vyjadřovací schopnost
  - příjemné na používání
- formálně ověřit tuto „spustitelnou specifikaci“
- produkční kód vygenerovat

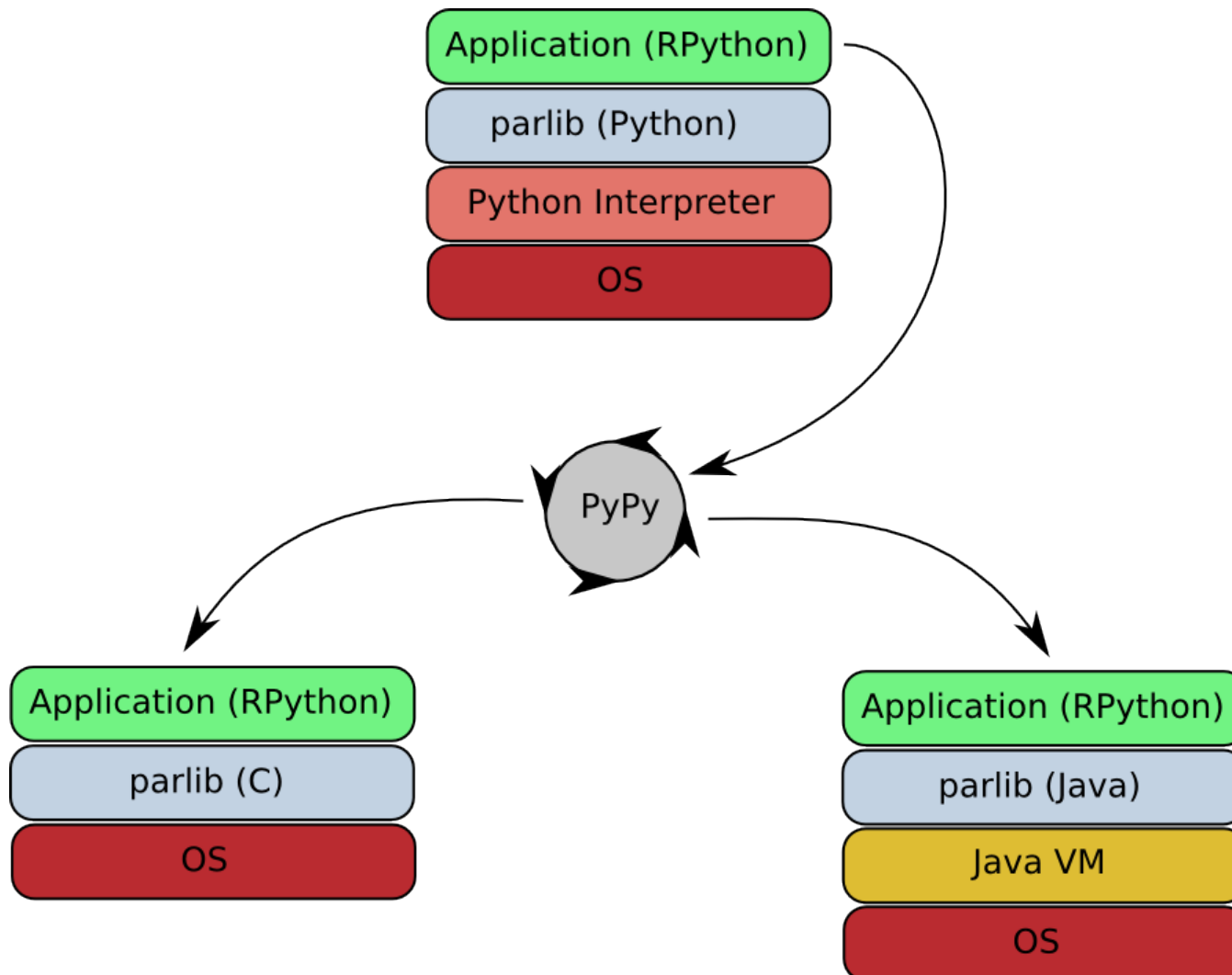
# RPython

- podmnožina jazyka Python
  - příjemné programování
- výsledek projekty PyPy (ETH Zürich)
  - experimentální interpreter a překladač Pythonu
- dobré vlastnosti dynamicky typovaných jazyků
  - krátký kód (méně chyb)
  - otevřeno pro nová paradigmatata (DbC, AOP)
- překlad do různých jiných kódů (C, JVM)

# Generování nativního kódu

- není možné použít interpret
  - dynamické vlastnosti jsou závislé na interpretaci
- řešení: po nějakou dobu dynamické chování, pak vygenerujeme nativní statický kód
- abstraktní interpretace, inference typů
- generování nízkourovňových aspektů (správa paměti, vlákna)

# Schéma generování

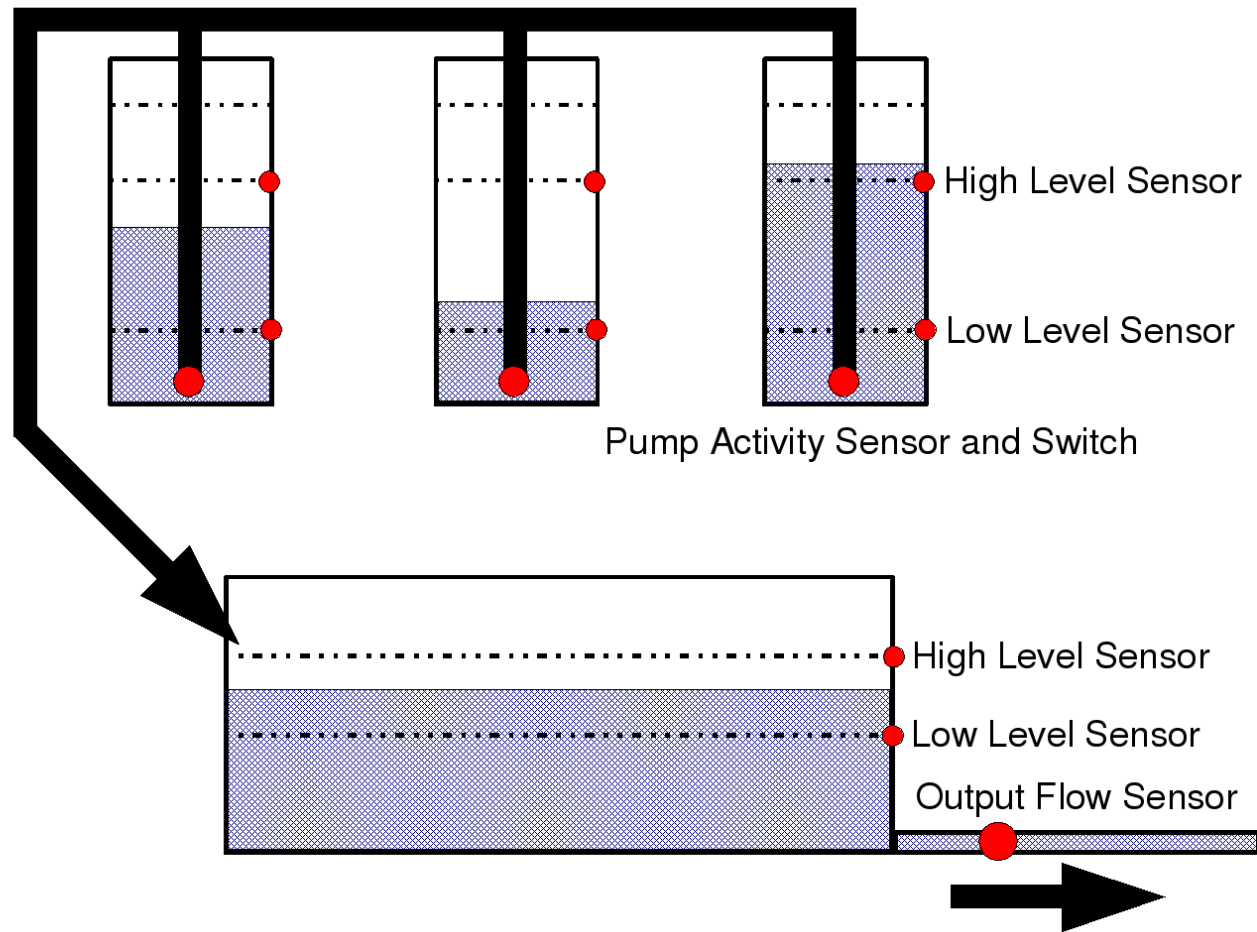


# Formální metody

- řada možností:
  - upravit PyPy pro překlad do Promely (SPIN)
  - udělat z PyPy interpreteru modelchecker
- Java PathFinder
  - modelchecker pro Java Bytecode
  - JVM s backtrackingem
  - jednoduché a funkční
- ...pokračování po případové studii...

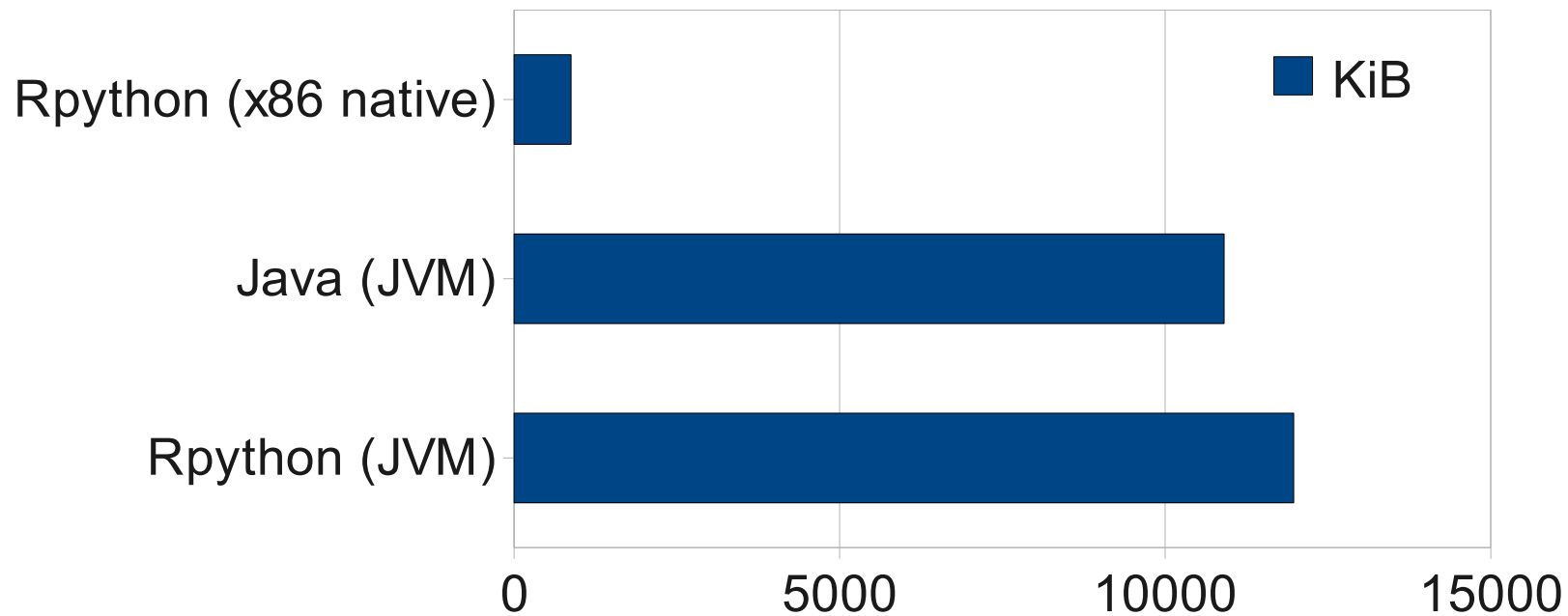
# Případová studie: vodárna

- každá nádrž = jedno vlákno
- omezení počtu běžících pump = synchronizace



# Případová studie - paměť

- Paměť obsazená procesem



# Ověřování s JPF

- vyhledávání uváznutí
- vyhledávání neošetřených výjimek
- dva způsoby použití
  - zkoumání části systému
  - procházení stavového prostoru celého systému
- spíše hodně důkladné testování než formální důkaz správnosti

Application (RPython)

parlib (Java)

JPF

Java VM

OS

# Záplata na souběh :-)

```
paskma@paskma: ~/projects/pypygit
File Edit View Terminal Tabs Help
paskma@paskma: ~/proje... x paskma@paskma: ~/proje... x paskma@paskma: ~/proje... x paskma@paskma: ~/proje... x
#
# a simplified version of the basic printing routines, for RPython programs
-class StdOutBuffer:
-   linebuf = []
-stdoutbuffer = StdOutBuffer()
+
def rpython_print_item(s):
-   buf = stdoutbuffer.linebuf
-   for c in s:
-       buf.append(c)
-       buf.append(' ')
-def rpython_print_newline():
-   buf = stdoutbuffer.linebuf
-   if buf:
-       buf[-1] = '\n'
-       s = ''.join(buf)
-       del buf[:]
-   else:
-       s = '\n'
import os
os.write(1, s)
+
+def rpython_print_newline():
+   import os
+   os.write(1, "\n")
#
compiled_funcs = FunctionCache()
commit e1b2c132dfcf9f8825b2fc9fafed4e06cd11d59c
:
```

# Průzkum stavového prostoru

- 5 vláken (3 model, 2 řídicí systém)
- 2 iterace každého cyklu
- procházení trvalo 8h 34m
- spotřebováno 787 MiB paměti
- nenalezeny žádné chyby

# Konec

- děkuji za pozornost