An Approach to Embedded System Development Based on Dynamically-typed Language

Marek Paška
Outline

1) Introduction (Embedded devices)
2) The proposed development process
3) Details of the compilation process
4) Case study [optional]
Software in Embedded Systems

• Constrained hardware resources (cheap HW)
• Dependable
  – failure may have severe consequences
  – hard to fix the errors
• Usually works in reactive mode
  – hard/soft realtime
• Computational time is more expensive than programmer's time
State of the Art in Embedded SW

- *(Development tends to be conservative)*
- Higher level, general purpose languages
  - Java
- Formal methods
- Model-driven development, generative programming, ...
Ease development even further?

- Python – very high-level language
  - generate efficient native code?
  - formal verification?
RPython

- Rich enough subset of Python
  - comfortable for programmer
- Part of the PyPy project (ETH Zürich)
  - experimental Python interpreter and compiler
- Good characteristics of dynamic languages
  - shorter code (less errors)
  - open for new paradigms (DbC, AOP)
- Translation to various codes (C, JVM)
Development Process

- Software is primary written in RPython
  - can run on standard Python interpreter
- C code can be generated from the RPython source
  - results in high performance native code
- Java byte-code is also generated
  - to be verified by tools developed for Java
Code Generation Scheme
PyPy Compilation Chain

- Object Space
  - abstract interpretation
  - initialization

- Flow Graph
  - type inference, transformations

- Flow Graph
  - source code generation

- Source Code

- Output Code
Abstract Interpretation

- **Input**: initialized graph of objects ("object space") in the memory of Python interpreter
  - And the selected entry point
- **Output**: internal PyPy program representation called *flow graph*
- data types of the initial flow graph are *abstract*
Flow Graph Example

[start block]

IN = <var_n1>

var_lesser1 = lt(var_n1, 0)
var_lesser2 = is_true(var_lesser1)

EXIT = {var_lesser2}

exitcase = False
arg = <var_n, 1>

IN = <var_n2, var_result1>
var_greater1 = gt(var_n2, 0)
var_greater2 = is_true(var_greater1)

EXIT = {var_greater2}

exitcase = True
arg = <>

IN = <>

var_exc_inst1 = simple_call(type_ArithmeticError)
var_exc_type1 = type(var_exc_inst1)

EXIT = {}
Flow Graph Transformations

- Can change the structure of the graph
- Can add new information
- Examples:
  - Add type annotations for a particular code generator
  - Add reference counting for GC
C vs. Java-btcd. Generation

• Java bytecode:
  - Assign JVM types to the abstract types
  - Generate bytecode

• C:
  - Assign C types to the abstract types
  - Exception transformation
  - GC transformation (empty for BoehmGC)
  - Generate C code
Java Pathfinder (JPF)

- Explicit model-checker for Java bytecode
- JVM with backtracking
  - deadlocks
  - uncaught exceptions
  - Linear Temporal Logic
Process Dependability

- How can we know that the C and Java bytecode programs behave the same?
  - The most tricky parts (object space initialization, abstract interpretation) are shared
  - We have precise definition of the additional transformation for the C compilation, however no formal proof of correctness
Shared Threading Model

- All variants of the program (interpreted RPython, C, Java bytecode) use *monitors* as we know them from the Java world
  - Allows JPF to perform optimizations
  - Monitors for C and RPython are implemented in the *parlib* library
  - *(They are structured and that's nice)*
Memory Requirements
Computational Performance

Numerical Polynom Integration

- PyPy-C backendopt -O6
- C -O6
- C
- PyPy-JVM
- PyPy-C

Time [s]
Linear Temporal Logic

- Defined over sequences of states
  \[ s_0, s_1, s_2, s_3, \ldots \]

- There are propositions that hold (not hold) for every particular state
  - \( \varphi = x > 4 \)
  - \( \varphi(s_2) = \text{true} \) or \( \varphi(s_2) = \text{false} \)

- Temporal operators
  - \( X\varphi, G\varphi, F\varphi, \varphi_1 U\varphi_2 \)
LTL Examples

• F(all_records_processed)
  - some positive event guaranteed

• G(there_is_at_least_oneRunnable_thread)
  - program is deadlock-free

• G(request => X(F(response)))
  - request is inevitable followed by response

• G(¬file_closed U result_written)
  - write and then close the file
Case Study: NVR

- Network Video Recorder is a device that manages IP cameras over computer network.
  - records video produced by cameras
  - records events produced by cameras
    - motion detection
    - alarms
NVR Internals

- For every camera there is a dedicated camera driver that downloads the video and events.
- Events are summarized to time intervals and then written into a database.
NVR Scheme
Real LTL Formula

- Whenever camera driver detects an alarm it is inevitably written into the alarm log.

\[ G((\text{method:Driver.alarmOccurred}) \rightarrow (X(F(\text{method:AlarmLog.writeAlarm})))) \]
Real LTL Formula (2)

- Whenever a time interval elapses, the summarized value is not cleared until it is written into the database

\[ G((\text{method:Summarizer.intervalElapsed}) \implies (X((\neg(\text{method:Summarizer.clearInterval})) \lor (\text{U(method:Database.writeInterval)}))))) \]
Conclusion

- A novel approach to embedded systems development
  - very high level description (RPython)
  - flexible code generation
  - LTL-properties verified by Java Pathfinder holds also for the production C code
The End

• Thank you for your attention.

• ?