Design, Implementation and Evaluation of the

Resilient Smalltalk

Embedded Platform

Mads Torgersen

University of Aarhus

Joint work by

Esmertec OOVM:

- Lars Bak
- Steffen Grarup
- Jakob Roland Andersen
- Kasper Verdich Lund

Aarhus University:

- Toke Eskildsen
- Klaus Marius Hansen
- Mads Torgersen

The drill of small embedded systems

- Low-level unsafe language
- Platform-dependent semantics
- Expensive code-run cycle
- Debugging requires special configuration or even hardware
- No post-deployment serviceability

We would prefer...

- High level, safe language
- Platform independence
- Complete code/run integration
- Full, continuous inspection/modification
- ... also after deployment

...please!

We would prefer...

- Hig
- Plat
- Cor
- Ful
- ... a

In practice...

Smalltalk!

...please!

But...

- Resource constrained devices
- Real-time demands
- No GUI

...not so Smalltalk.

- Resilient: How small can we get but stay nice?

The Resilient Platform

- No OS (Smalltalk back in charge!)
- External programming environment through reflective interface (no self modification)
- Small interpreter-based VM (no native compilation)
- Real-time GC
- Eat our own dog-food (scheduler, tcp/ip stack etc. written in Smalltalk)
- Some language changes

Programming environment

- Eclipse plug-in with network connection to running device
- Editor, byte-code compiler, debugger
- Source code based!
- Keeps source and device synchronised via reflective interface
- Designed towards a ”Smalltalk experience”
- Can be connected to VM at any stage

The Resilient programming language

- Several deviations from standard Smalltalk
  - compliance not a priority
- Source code based – syntax for classes
- Low-level synchronisation construct test-and-set
- LIFO blocks – statically typed!
- Evaluation order ”backwards”
- Lexical namespaces

Example

```
Mutex = Object (
   | owner |
   do: [action] = (
     [ owner ? nil := Scheduler current ]
       whileFalse: [ Scheduler yield ].
     action value.
     owner := nil
   )
)
```

Typed LIFO blocks

- Surviving blocks
  - performance problem
  - rarely used
- Resilient: LIFO (last-in-first-out) behaviour
  - compiler enforced
  - ” type declaration”: `do: [action] = ( ... )`
  - cannot be assigned or returned

Typed LIFO blocks

- Pros
  - No heap allocated environments: significant performance boost
  - No stress on GC
  - Safe non-local return
- Cons
  - Loss of purity (two static types, not one)
  - Comparators
  - GUI callbacks

Virtual machine

- Interpretation – no compilation
  - space for speed
  - no bridges burnt
- Everything in the heap
  - embedded memory too small for segmentation
  - growable stacks
  - bytecode for methods and blocks
  - everything subject to GC

Virtual machine

- Real time GC
- Philosophy: Small is better than fast, but small is also often fast
  - footprint < 64K
  - realistic applications comfortable in 128K
  - fast, but not as fast as compiled code
- Safe memory access

```
io := Memory at: 16r90040000 size: 16r20
```

Conclusions

- Small Smalltalk is for real
- Unorthodox approach helps
- Comfortable niche between C/assembler and Java CLDC
- Embedded programming for ordinary programmers
- Open issues
  - deployment
  - performance-critical code

Oh, by the way...

TM ® © TM ® © TM ® © TM ® © TM ® © TM
® © TM ® © TM ® © TM ® © TM ® © TM ®
© TM ® © TM ® © TM ® © TM ® © TM ® ©
TM ® © TM ® © TM ® © TM ® © TM ® © TM
® © TM ® © TM ® © TM ® © TM ® © TM ®
© TM ® © TM ® © TM ® © TM ® © TM ® ©
TM ® © TM ® © TM ® © TM ® © TM ® © TM
® © TM ® © TM ® © TM ® © TM ® © TM ®
© TM ® © TM ® © TM ® © TM ® © TM ® ©
TM ® © TM ® © TM ® © TM ® © TM ® © TM

Thank you!™