“Formal Development vs. Formal Verification

The OpenComRTOS Example"

Eric.Verhulst@altreonic.com

From Deep Space to Deep Sea

Altreonic

Content

- Why Formal Techniques are needed
- The holy grail of FM
- Formal verification of software verification?
- Formal modeling and formal verification
- SE dependencies
- OpenComRTOS project FM level 1 project
- Examples of FM lite
- Conclusion

The Wall of Complexity challenge(1)

Murphy‘s law:

- things going wrong is a matter of probability
- and the odds are getting worse

Focus domain: embedded systems

Products become systems:

- ‚Smart‘ by using 10‘s of processors
- Distributed operation
- Connected to other systems (incl. wireless)
- Human in the loop
- Requirements:
  - High quality, high reliability
  - High level of safety, fault-tolerance
  - Secure operation
- And as well:
  - Cost-efficient (life-cycle cost)
  - Competitive
  - Upgradeable

The Wall of Complexity challenge(2)

The solution is NOT to link the myriad of existing processors, software tools, etc.

Because they are <u>semantically</u> too different

Because we have too many of them

But none supports scalability requirement

But few support graceful degradation

Time to apply occam‘s razor:

- Back to basics
- Get rid of unnecessary complexity and historical ballast
- More engineering, less crafting, more scalability and real re-use

=> ‚Trustworthy Embedded Components‘

- Reliability, correctness
- Safety, fault-tolerance
- Security
- Formally developed and validated software & IP
- Open Technology License: source code + validation & design data

Why do we need formal techniques?

How precise is the engineer's brain?

How precise is the management's brain?

How precise can we define requirements?

How precise can we define specifications?

How precise can we « write » software?

How precisely do we know all dependencies?

How sure can we be of the end-result?

Can we trust our mind ?

How many « F » do you count ?

**FINISHED FILES ARE THE RE
SULT OF YEARS OF SCIENTIF-
IC STUDY COMBINED WITH
THE EXPERIENCE OF YEARS**

Can we trust our mind ?

How many « F » did you find ?

FINISHED FILES ARE THE RE SULT OF YEARS OF SCIENTIF- IC STUDY COMBINED WITH THE EXPERIENCE OF YEARS

Did you see the similarity with source code (debugging) ?

The holy grail of FM

A verifying compiler uses automated mathematical and logical reasoning methods to **check the correctness of the programs that it compiles**. The criterion of correctness is **specified by types, assertions, and other redundant annotations** that are associated with the code of the program, often inferred automatically, and increasingly often supplied by the original programmer. The compiler will work in combination **with other program development and testing tools**, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components. ….

An important and integral part of the project proposal is to evaluate the capabilities and performance of the verifying compiler by **application to** a representative selection of **legacy code**, chiefly from open sources. This will give confidence that the engineering compromises that are necessary in such an ambitious project have not damaged its ability to deal with **real programs written by real programmers**. It is only after this demonstration of capability that programmers working on new projects will gain the confidence to exploit verification technology in new projects.

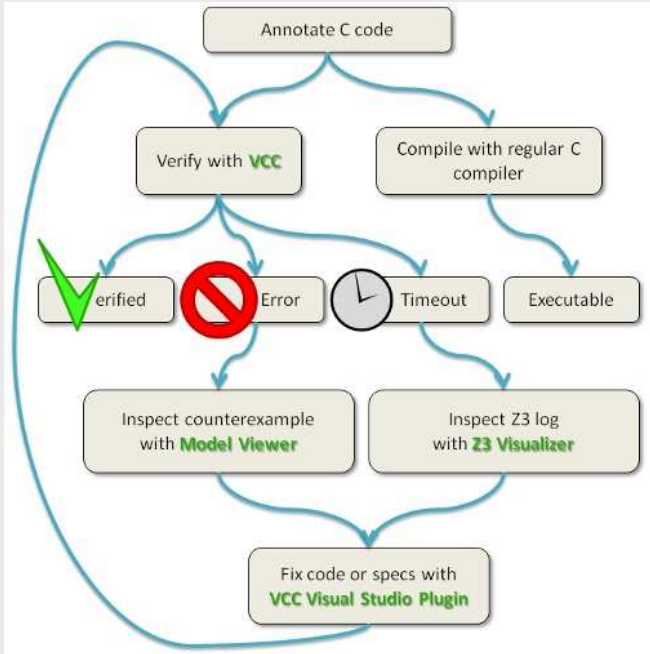
Note that **the verifying compiler itself does not itself have to be verified**. It is adequate to rely on the **normal engineering judgment** that **errors in a user program are unlikely to be compensated by errors in the compiler**.

The verifying Compiler: A Grand Challenge for Computing Research [Hoare, 2003]

VCC: A Verifier for Concurrent C

VCC is a tool that proves correctness of annotated concurrent C programs or finds problems in them. VCC extends C with design by contract features, like pre- and postcondition as well as type invariants. Annotated programs are translated to logical formulas using the Boogie tool, which passes them to an automated SMT solver Z3 to check their validity.

(approach used by MS for verification of Hypervisor)



Hypervisor example

What is being verified:

- 60 000 lines of source code
- C + x64 assembly
- Properties
  - Safety: Basic memory safety
  - Security: OS Isolation
  - Utility: Hypervisor services guest OS with available resources

- About 3 functions per day are formally verified
- Input is C source, but of course, errors have been found
- But:
  - **what if a new release of the Hypervisor is made?**
  - **what if I change one line of code?**

The question

IF we have 1 million lines of formally verified source code, what can we do with it?

IF the Hypervisor source code is 100% formally verified, what will have been proven?

Answer:

- We will have proof that the source – as it is – is correct:
  - Each piece on itself
  - IF we don't change anything
- What about: concurrency? architecture? efficiency? reuseability? scalability?

Formal techniques

In computer science and software engineering, **formal methods** are a particular kind of mathematically-based techniques for the **specification, development and verification** of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can **contribute to the reliability and robustness** of a design.

However, the high cost of using formal methods means that they are usually only used in the development of high-integrity systems, where safety or security is of utmost importance. (src: wikipedia)

Levels of FM

Level 0: Formal specification may be undertaken and then a program developed from this informally. This has been dubbed *formal methods lite*. This may be the most cost-effective option in many cases.

Level 1: Formal development and **formal verification** may be used to produce a program in a more formal manner. For example, proofs of properties or **refinement** from the **specification** to a program may be undertaken. This may be most appropriate in high-integrity systems involving safety or security.

Level 2: Theorem provers may be used to undertake fully formal machine-checked proofs. This can be very expensive and is only practically worthwhile if the cost of mistakes is extremely high (e.g., in critical parts of microprocessor design).

<u>Practice:</u>

- Formalised specification / design /development
- Formal model checkers (automated once model is build)
- Formal provers (often manual)

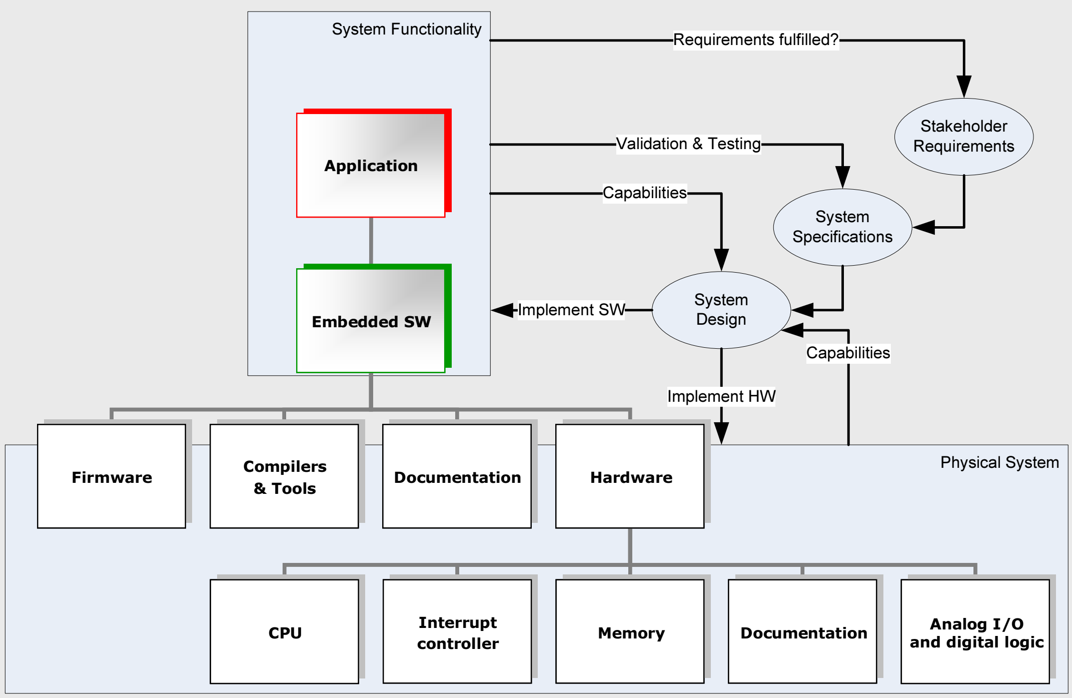
Where do formal techniques fit in?

- Engineering without using mathematics as tool is not engineering, it is craft-ing of art-ing
- The issue in HW and SW design is complexity:
  - Discrete domain vs continuous domain
  - State space explodes
    - combinatorially (HW)
    - exponentially (SW)
    - no graceful degradation
- FM help to reduce the state space and to prove the absence of undesired states
- FM support engineering process, but don’t replace it as a human activity

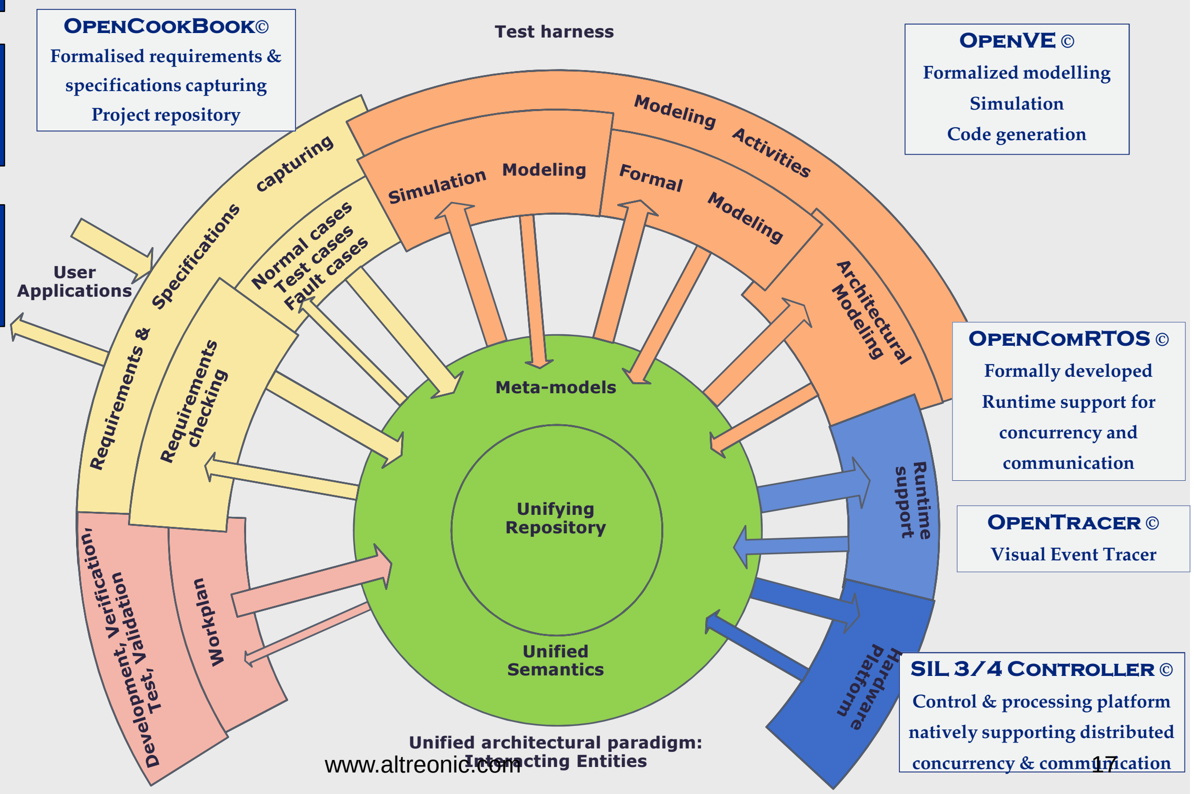
What systems engineering process?

- Do we develop the right system? (1)
- Do we develop the system right? (2)
  - (1) is intentional domain => stakeholders
  - (2) is extensional domain => process view
- What is good engineering?
  - Proven heuristics
  - Elegance in design
  - KISS: keep it simple and smart:
    - A complex solution is a problem not well understood. A simple solution required hard thinking.
      - This is were Formal Techniques can help

SE Process dependency graph



Unified Systems/Software engineering



Formal modeling Level 1
for developing OpenComRTOS

- Funded R&D project (IWT, Flanders)
  - Open License Society: technology development
  - University Gent (INTEC, Prof. Boute): formal modeling
  - Melexis: co-sponsor and first user (16bit uC)
- GUI tools:
  - graphical modeling/development environment
- Goal:
  - Develop Trustworthy <u>distributed</u> RTOS
    - Follow OLS SE methodology
    - Formal verification & analysis: formal modelling
  - Scalable distributed RTOS
  - Verify benefits and issues of using Formal Modeling

Formal modeling tools

Default mathematical approach:

- Correctness by proof
  - Labor and time intensive
  - Needs specialists
  - (Human) Error prone process
- Tools needed
  - State space is exponentially large
  - Issues always in « hidden corners »
  - Allow incremental process
- Requirements:
  - Support state machines
  - Support concurrency and communication
  - Low notational barrier

Formal modeling tools: selected options

Investigated:

- SPIN, B, CSP (FDR), **TLA+/TLC**

Outcome of process:

- SPIN OK, initially preferred, good documentation, wide user base, but very C-like style
- CSP: hard notation, FDR not readily available
- B: waiting for Event B, incremental approach and compositionality very good
- TLA+/TLC
  - Based on Temporal Logic
  - Mathematical notation, but standard
  - Works for any domain (SW, HW, …)

Benefits of TLA+/TLC

- TLA+/TLC home page on http://research.microsoft.com/users/lamport/tla/tla.html
- Initial models reflected “programming style”
  - That’s the way the mind works (after being conditioned …)
  - > 28 successive models from 2 pages to 25 pages
    - Initially very abstract, neglecting details
    - All successive models were correct, why ?
      - Iterative, incremental process!
      - Takes 15 minutes from one model to the next
    - Interplay between software architects and formal modeling engineer
      - Architectural model polluted by programming concepts
      - Abstraction from TLA helped to find these issues
      - Result: much cleaner, safer and performant architecture
- TLA models do not prove software is correct (! ?)
  - TLC proves that Formal **Models** are correct

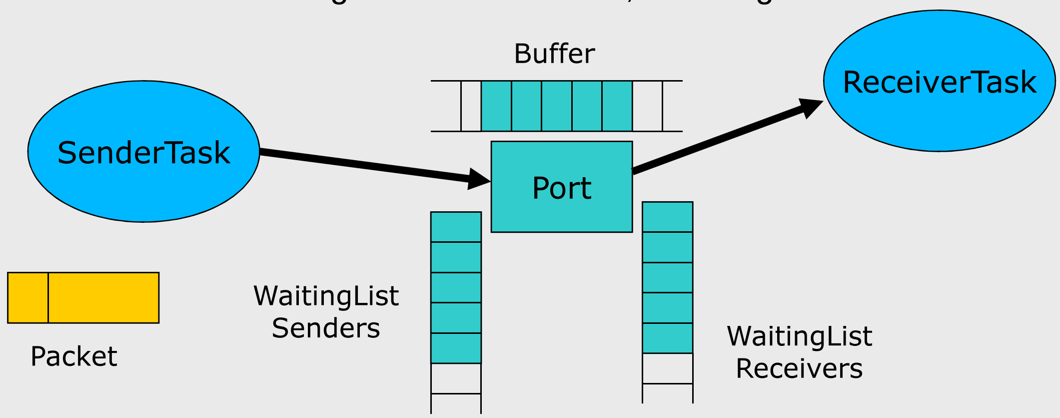
Issues with TLA+/TLC

- Needs a few months to get the right modeling style (especially concurrency)
- TLC declares critical section over all actions
  - In RTOS must be minimal
  - Requires good know-how of target processor
  - Why can’t FM not give the minimum critical sections?
- State Space is exponential
  - Millions of states for small application test model
  - Might need hours to check
  - Tracing illegal states not always trivial
- => For time-out behaviour we used UPPAAL

Example

OpenComRTOS entities:

- Tasks (= function with its own workspace and context)
- Ports: for exchanging L0_Packets
  - SendPacket_(N)W(T)(async)
  - ReceivePacket _(N)W(T)(async)
  - Needs: waiting lists on both sides, buffering



Formally modeled

```
TypeInvariant  == /\ ppool \in [Adr-> Packet \union {NoData}]
                  /\ PQ \in [ FIFO : [Port -> Seq(Adr)],
                              WL   : [Port -> Seq(Adr)]]
                 /\ chan \in [ val: [HLink -> Packet \union {NoData}],
                               stt: [HLink -> {"free","busy"}]]
                  /\ TxQ \in [TxChan -> Seq(Packet)]
                  \*    /\ tstate \in [UTask ->{"running","ready","wait4anS","wait4anR"}]
```

$$TypeInvariant \triangleq \land\ ppool \in [Adr \rightarrow Packet \cup \{NoData\}]$$

$$\land\ PQ \in [FIFO : [Port \rightarrow Seq(Adr)],\ WL : [Port \rightarrow Seq(Adr)]]$$

$$\land\ chan \in [\text{val: } [HLink \rightarrow Packet \cup \{NoData\}],\ \text{stt: } [HLink \rightarrow \{\text{"}free\text{"}, \text{"}busy\text{"}\}]]$$

$$\land\ TxQ \in [TxChan \rightarrow Seq(Packet)]$$

$$\land\ tstate \in [UTask \rightarrow \{\text{"}running\text{"}, \text{"}ready\text{"}, \text{"}wait4anS\text{"}, \text{"}wait4anR\text{"}\}]$$

What was wrong?

```
FIFO : [Port -> Seq(Adr)],
WL   : [Port -> Seq(Adr)]]
```

- Both (abstract) models are the same
- Natural language is imprecise, semantics are context driven
- We forget the hidden assumptions
  - FIFO := buffering of data
  - WaitingList := waiting, descheduled
  - But: both « buffer » and « wait »
  - Result: either FIFO, either WaitingList
  - « Trick »: all entries are pointers (Addr) to Packets
- Benefits:
  - Infinite buffering until no more memory (for Packets)
  - Overflow-free buffering

Other example

- OpenComRTOS layer L1
  - Traditional RTOS support:
    - Preemptive scheduling with priority inheritance support
  - Services with rich and diverse semantics:
    - Events, semaphores, FIFO-queues, mailboxes, channels, pipes, mutex, resources, memory pools, …
    - « distributed semantics »
  - 100’s of RTOS with such support
  - 15 years of experience, 3 generations of RTOS design
  - L1 layer can be any API
- What did we find?
  - Scheduling algorithm can be improved to reduce worst-case rescheduling latency and blocking time
  - All RTOS objects are variations of the same generic « hub » object. Result: less but faster code
    - 5 KBytes vs. 50 KBytes before

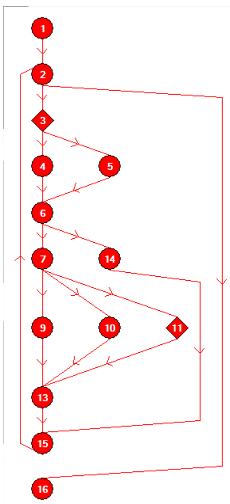
Impact on code quality

RTOS code is often labeled as a ‘black art’

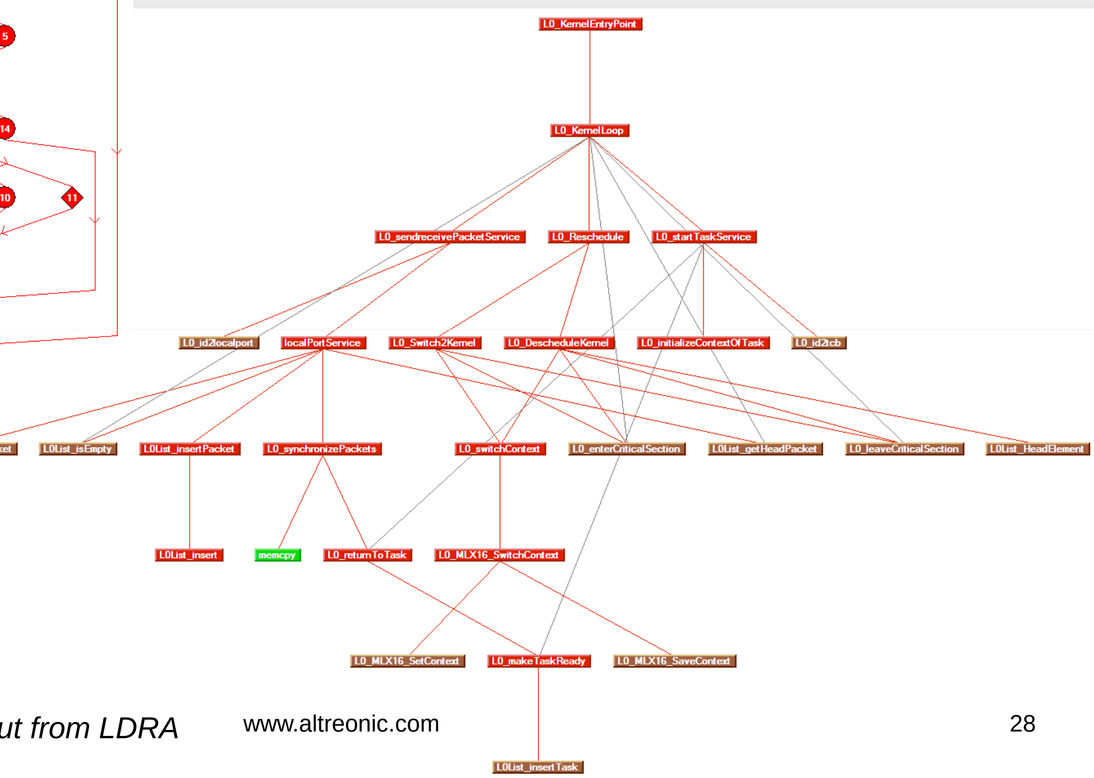
- CPU dependency
- Assembler for speed
- Asynchronous operation and jump tables

Our results (fully in ANSI-C, Misra-C checked)

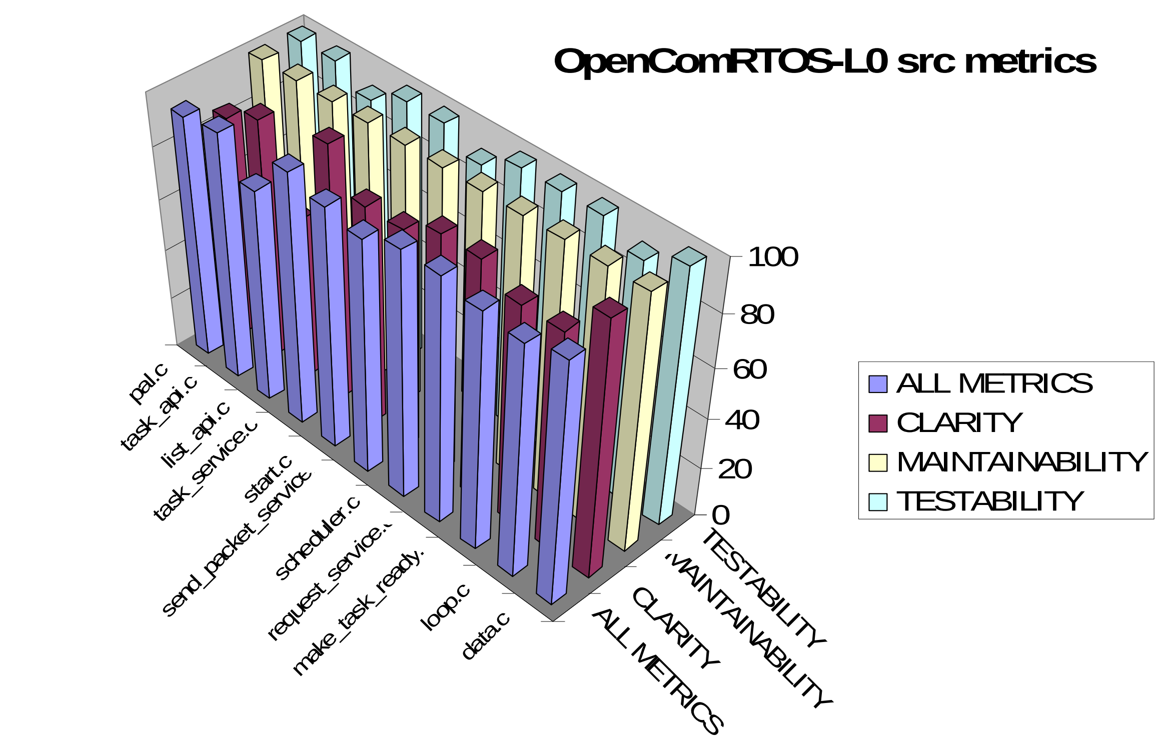
- SP(L0): < 1000 machine instructions
- MP(L0): < 2000 machine instructions
- Needs a few 100 bytes of data RAM
- Very portable and maintainable
- Runs on MelexCM (16 bit) and Windows



OpenComRTOS Kernel structure



Output from LDRA



Output from LDRA

Examples of FM Lite (1)

Safe Virtual Machine for C

- Instruction set (IS) and its behaviour defined in Haskell + safety extensions
- Haskel used to write VM generator (generic for IS!)
- VM is generated as C program
- C code is never touched and can be compiled to any target
- Issues related to native vs generic data representation
  - Byte order! + side effects at bit level
- Result (IS = ARM Thumb1):
  - SVM Program Code: 3.8 Kbytes
  - Data requirements: 500 bytes

Examples of FM Lite (2)

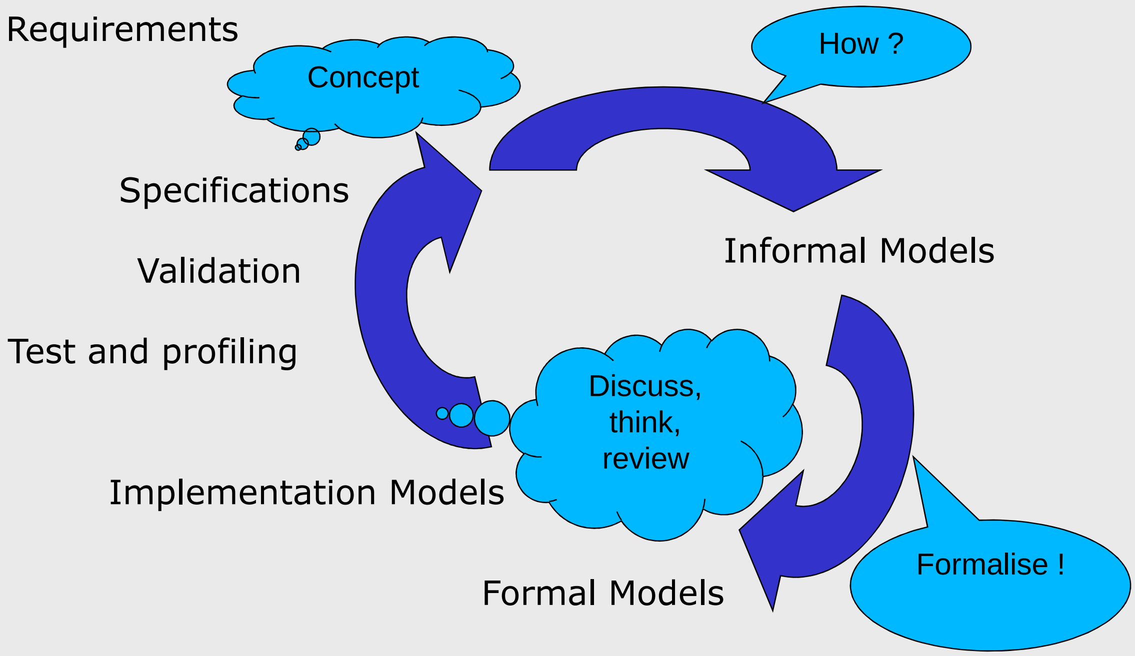
CoolFlux DSP core

- Instruction set and its behaviour defined in nML
- Target Compiler (retarget.com) tools generate:
  - VHDL of CPU, C compiler (no optimisation), simulator, debugger
- Close link between CPU architecture and C compiler
- Allows to play “what-if” scenarios
  - Optimisation for DSP performance, size, power consumption
- CoolFlux: PMEM (32b), XMEM (24b), YMEM(24b)
- BSP variant: can handle complex 12bit numbers
- Result of porting OpenComRTOS (SP):
  - Prog Code: 2 Kwords
  - Data requirements: starts at 750 bytes
  - Complete port only took 2 weeks (on simulator)

Conclusions

- Formal techniques are not THE solution
- But they are essential supporting techniques for the SE process that is a human driven process
- Formal verification without formal modeling has limited useability as it says nothing about the architecture
- Formal modeling is very powerful in finding better solutions and can be more efficient than using only formal verification. Formalisation is a first step.
- Contact: Eric.Verhulst @ altreonic.com

How it really works: teamwork



From theoretical concept to products

From Deep Space to Deep Sea

Altreonic

*"If it doesn't work, it must be art.
If it does, it was real engineering"*