

Towards Certified Compilation for IEC 61499

Per Lindgren and Marcus Lindner
Luleå University of Technology

**David Pereira and Luís Miguel
Pinho**
CISTER / INESC TEC, ISEP

2016-09-06

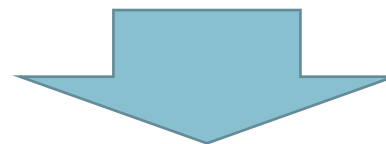




Goal: verified software

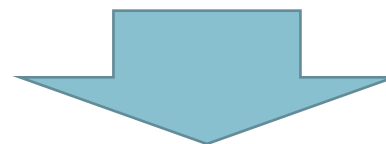
Rationale

Correct programming is hard
*Also **tools** may contain bugs*



Our research

Provide means for programmers to facilitate *correct software* development in the context of industrial automation

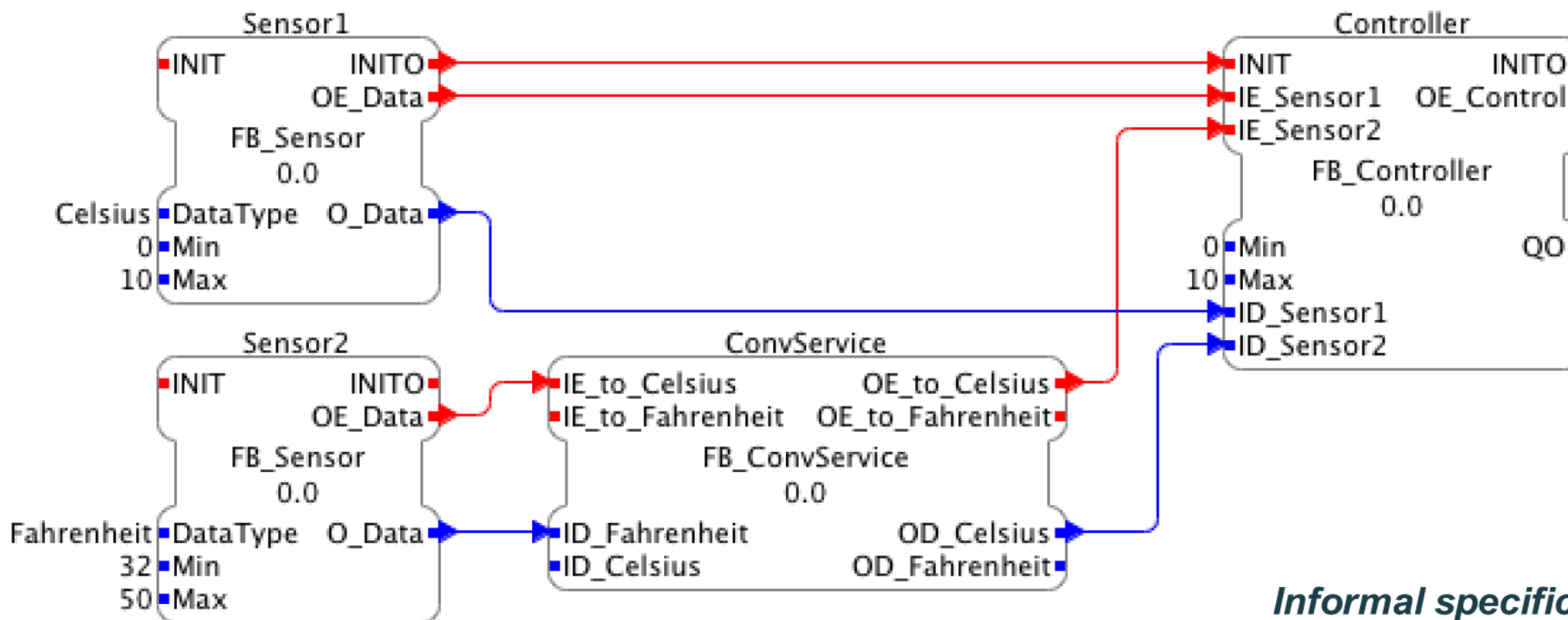


Project objective

“Improving the *robust operation of variable-frequency drives* in energy production plants in terms of software”



INDIN 2016, Verification of IEC 61499 applications (first ideas was presented at ETFA 2015)

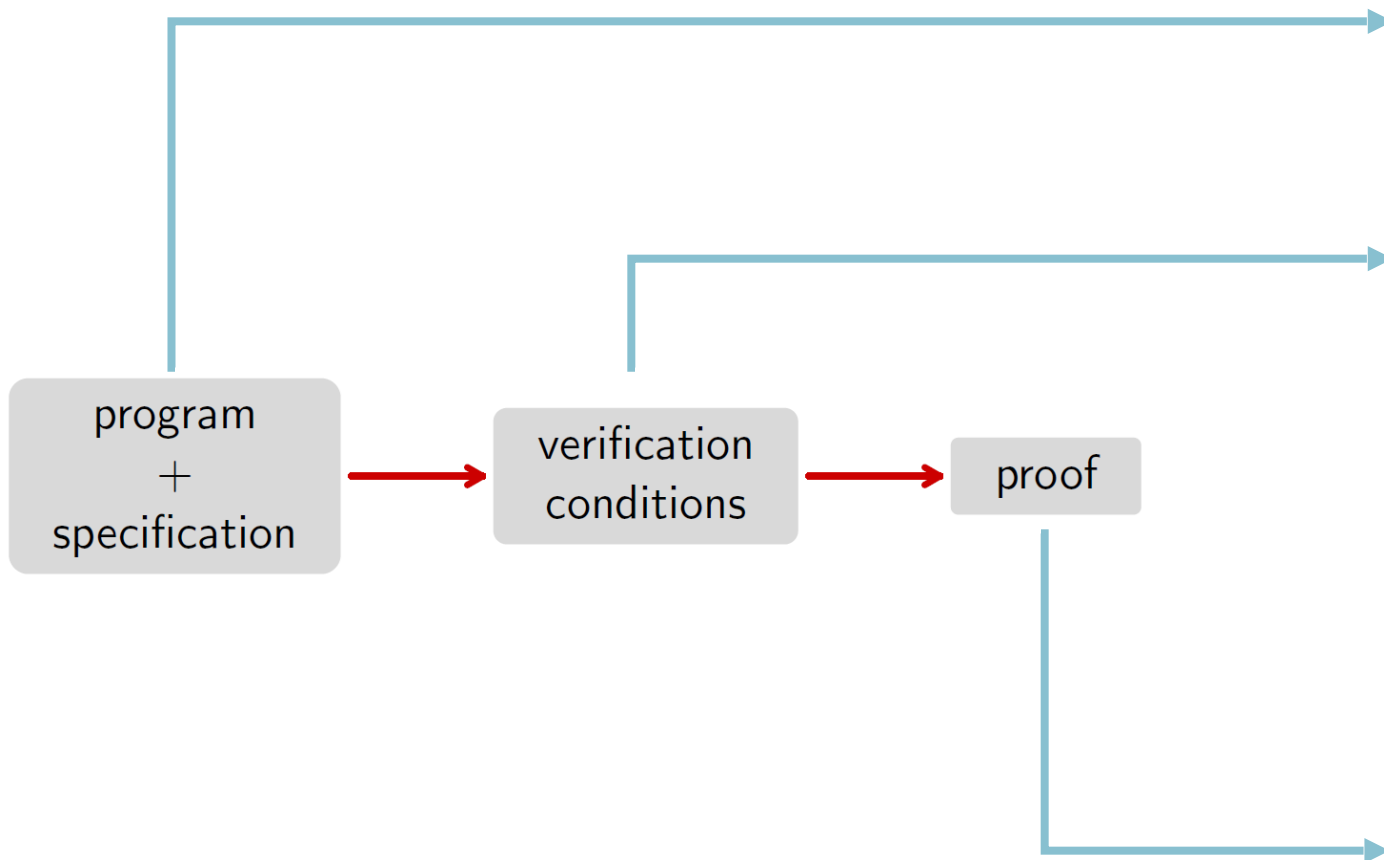


Informal specification:

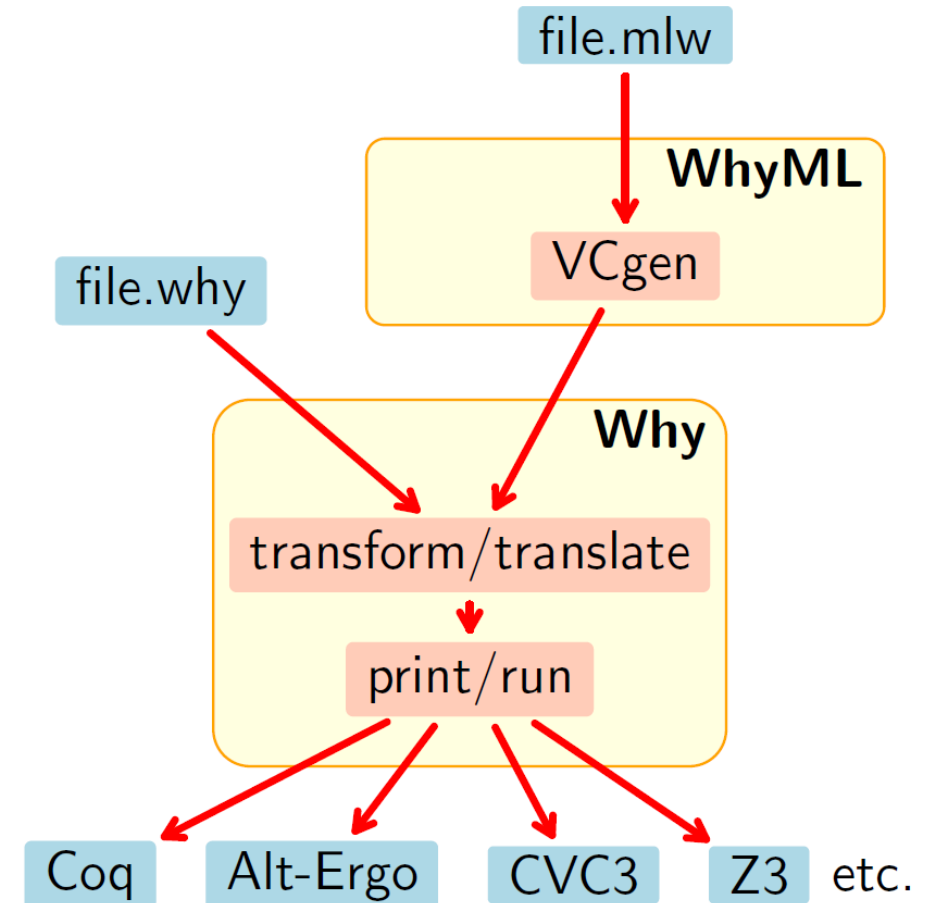
- 2 sensors
 - $S_1: 0^{\circ}C \leq c < 10^{\circ}C$
 - $S_2: 32^{\circ}F \leq f < 50^{\circ}F$
- 1 bang-bang controller (output: $S_1 < S_2$)
- Celsius-Fahrenheit conversion service

Why3: deductive program verification

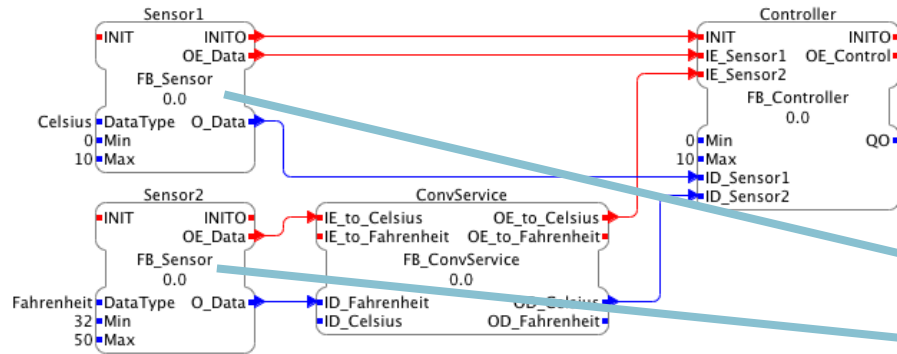
General principle




Why3 structure



Example: formalized specifications



Formal specification for
component interfaces
→ the contract!



```
module SensorGen
  use export Data

  constant min      : int
  constant max      : int
  constant metric   : metric_t

  type val_t

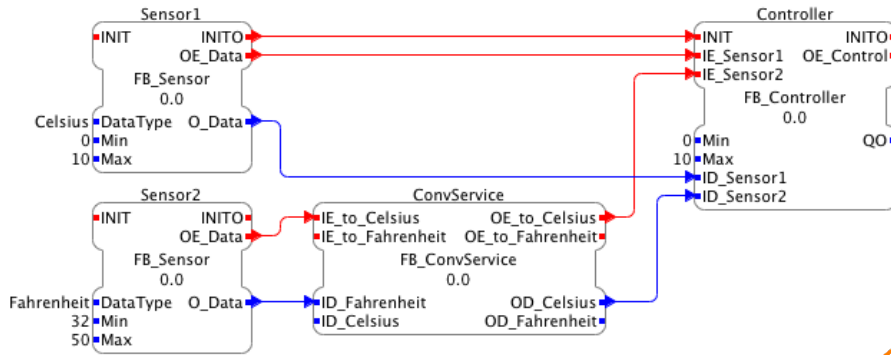
  predicate in_range (v : val_t) =
    min <= fst v < max /\ metric = snd v

  val read () : val_t
    ensures { in_range result }

  function range_of () : int = max - min

end
```

Example: system/FB composition



Verification of compositional soundness (w.r.t. the defined contracts)

```
module System
  clone import Sensor1 as S1
  clone import Sensor2 as S2

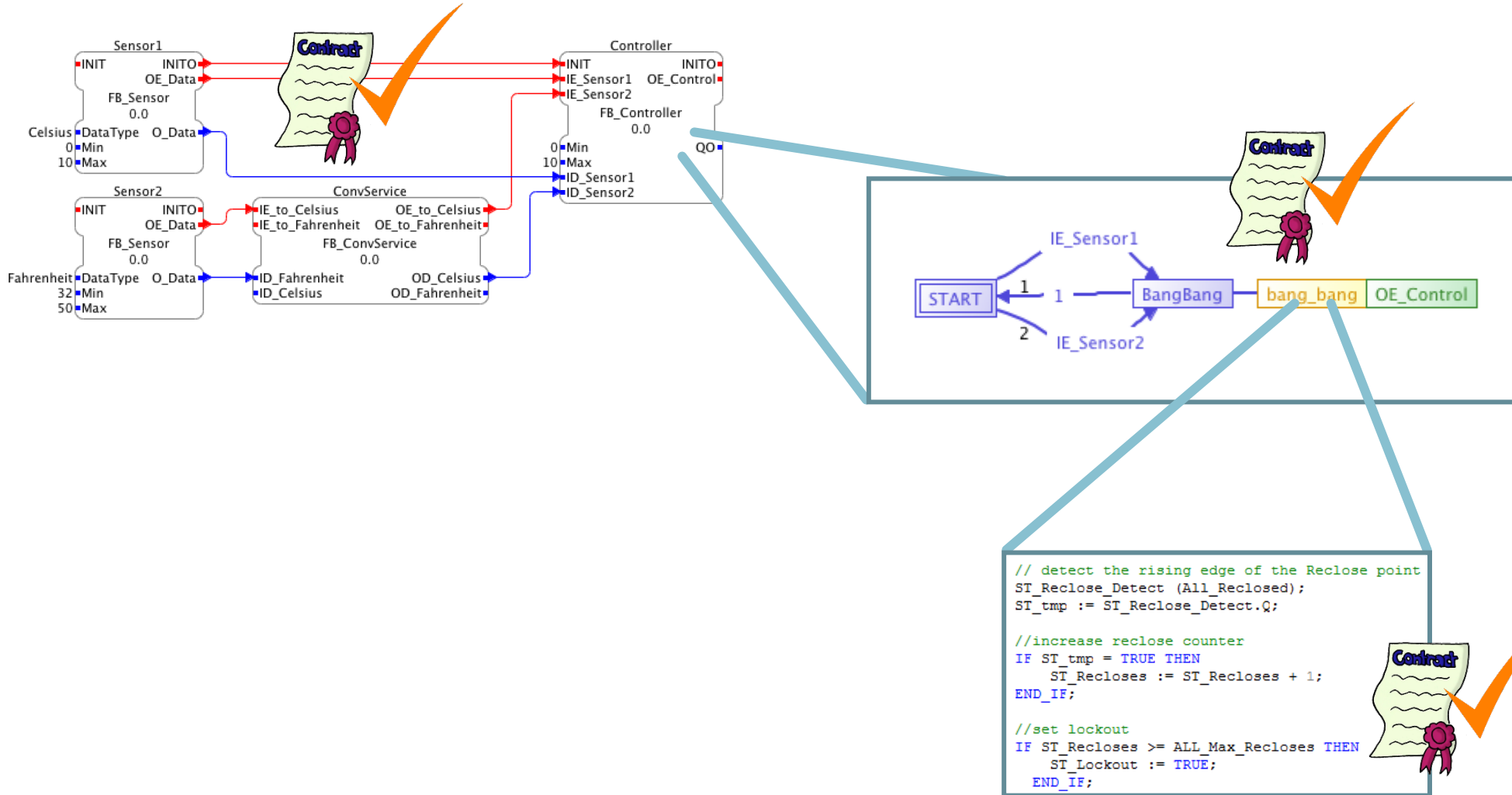
  (* should work in Celsius range 0 <= c < 10 *)
  predicate range (c:int) = 0 <= c < 10
  clone import ControlGen with predicate in_range = range

  let orchestration () =
    (* take readings from the sensors *)
    let s1_v = S1.read () in
    let s2_v = S2.read () in

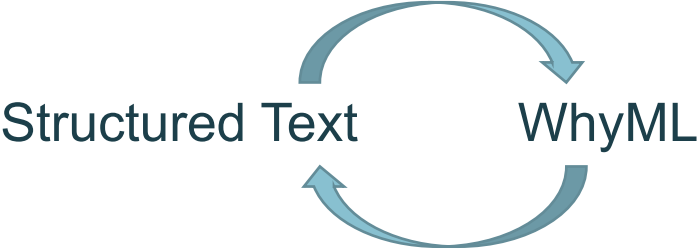
    (* present readings to the controller *)
    let bang = control s1_v s2_v in

    (* make sure the controller meets our expectations *)
    assert {
      match bang with
      | True  -> to_Celsius s1_v <  to_Celsius s2_v
      | False -> to_Celsius s1_v >= to_Celsius s2_v
      end
    }
  end
end
```

What's more



Future work towards verified implementations

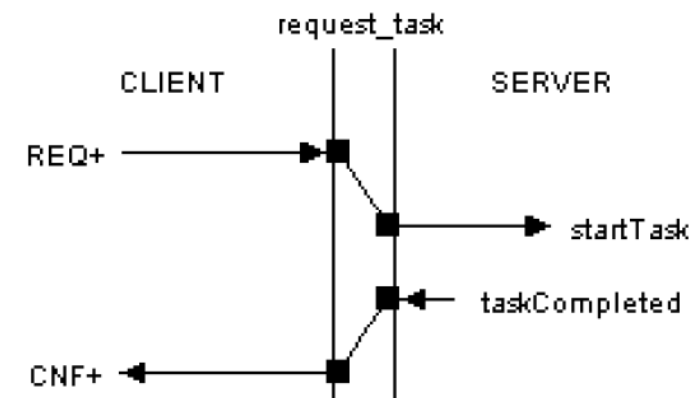
- Currently: Why3 encoding of IEC 61499 programs made by hand (should be largely automatized)
- suitable representations for the IEC 61499/IEC 61131-3 data types defined in WhyML and deployed for automatic translation
- Language translation:


```
graph LR; ST[Structured Text] --> WM[WhyML]; WM --> ST;
```
- Tool integration: Why3 platform directly accessible through Function Block IDE *4DIAC*



ETFA 2016, WIP, Further Verification Possibilities

- Deductive verification approach in contrast to model checking approach
- Limited possibilities of IEC 61499 to express behavior at component level: service sequences (no notions of timing, data, and state)



- design by contract approach is proof enabling and allows for the extraction of verified implementations



ETFA 2016, WIP Towards Certified Compilation of RTFM-core Applications

- RTFM-core : Concurrent tasks with **nested** critical sections
- Critical sections protected by named resources
- Deadlock free execution onto single core MCUs under the Stack Resource Policy
- **Mapping from IEC 61499 to RTFM-core possible** (C/C++ for algorithms)
- **CoreCert Certified Compiler** that
 - flattens nested structure into flat sequence of instructions
 - generates ceilings for named resources
 - proven correct to semantic model(s) in Why3
 - extracted to OCaml to generate executable

Towards Certified Compilation of RTFM-core Applications
 Per Lindgren¹ Marcus Lindner² David Pereira² Luis Miguel Pinho²
¹Luleå University of Technology Email: {per.lindgren, marcus.lindner}@lulea.se
²CISTER / INESC TEC / ISEP Email: {dmpre, lmp}@isep.ipp.pt

Abstract
 Concurrent programming is dominated by thread based solutions with lock based critical sections. Careful attention has to be paid to avoid race and deadlock conditions. Real Time for the Masses (RTFM) takes an alternative language approach, introducing tasks and named critical sections (no resources) solely in the RTFM-core language. RTFM-core programs can be compiled to native C/C++ and efficiently executed onto single-core platforms under the Stack Resource Policy (SRP) by the RTFM-kernel. In this paper we formally define the well-formedness criteria for SRP based resource management, and describe a certified (formally proven) implementation of the corresponding compilation from nested critical sections of the input RTFM-core program to a resulting flat sequence of primitive operations and scheduling primitives. Moreover we formalize the properties for resource ceilings under SRP and develop a certified algorithm for their computation.

RTFM-core Language and Model of Computation
 The RTFM-core language [1] allows reactive real-time systems to be specified in terms of time constrained, parametrised tasks with (potentially nested) critical sections protected by named resources. Whereas the minimalist RTFM-core language focus on concurrency limited C code is used to specify the functionality.

RTFM-core Grammar (Simplified)
 Type ::= #; CCode <#> Task let M [Smem] | Top Typ
 Stmt ::= #> CCode <#> claim M [Smem] | Stmt Stmt
Figure: Simplified grammar of the RTFM-core language

RTFM-core Example

```

Task 1 T1 {
  claim R1 { #> #> x:=x+y; #<#>
}
Task 1 T2 {
  #> #> #<#>
  claim R2 { #> x:=x; y:=x; #<#>
}
Task 3 T3 {
  claim R2 { #> #> #<#>
}

```

Listing 1 depicts a system with tree tasks T1, T2 and T3 with priorities 2, 1, 3 respectively. The resource R1 ensures race free access to x between tasks T1 and T2, while R2 ensures consistency of the value v for the update of x in T2. Access to y is race free under the assumption that y is atomic.

RTFM-kernel
 Targeting lightweight micro-controllers, the RTFM-kernel implements the scheduling primitives in terms of C-monads exploiting the underlying bare-metal storage hardware for single core, static priority, preemptive scheduling under the Stack Resource Policy (SRP). SRP based scheduling [2] brings benefits such as deadlock free execution and single blocking for single core, fixed priority preemptive scheduling. Alternative implementations of the scheduling API are available for multi- and many-core systems through thread bindings [6].

RTFM-core Compiler
 The RTFM-core compiler translates the RTFM-core model into plain C code, with inline references to the scheduling primitives. In the compilation, the nested critical sections are transformed to flat code sequences. Targeting the RTFM-kernel, static priorities are derived for the time constrained tasks, and static resource ceilings are derived for the critical sections according to the requirements of SRP.

Why3 – a Platform for Deductive Program Verification
 Why3 is a program verification platform that follows along the principles of deductive program verification established by Floyd [3] and Hoare [6]. It provides the language WhyML, designed with the purpose of serving as an intermediate verification language providing a rich many-sorted First Order Logic for specifying logical properties of the target developments, which are definable either via theories or via constructs that are coupled with function and type signatures. Among its several features, we can find in WhyML support for polymorphic records & (co)-inductive data types and predicates, and ghost code. WhyML, therefore allows to logically specify and implement programs. When specifications and implementations are given as input to Why3, the platform generates the set of proof obligations, i.e., the logical assertions that need to be proved correct in order to ensure that the implementations comply to their specifications. Furthermore, Why3 interfaces with a set of external theorem provers, both assisted and automatic, with the goal of automating the most possible the discharging of the generated proof obligations.

Why3 Data Types, for the RTFM-core compiler

```

type core_t =
  | 0 : c_stat_t
  | 1 : claim : real_t list core_t
type instr_t =
  | 0 : c_stat_t
  | 1 : lock : real_t
  | 2 : unlock : real_t
type t = core_t list
type r_t = instr_t list

```

Listing 2 depicts the AST for the task bodies of the input language core_t (RTFMspec) and the flattened sequence of instructions instr_t (RTFMkernel).

Certified Compilation
 In this following we will detail the contributions towards a certified RTFM-core compiler. We focus on two key aspects of the compilation process, namely the flattening of critical sections in the source program and the derivation of resource ceilings for scheduling by the SRP-based RTFM-kernel.

Correct compilation of RTFMspec
 We now introduce the conditions that determine the logical correctness of the compilation process from RTFMspec programs into RTFMkernel programs. Let r represent an RTFMspec program, and let r' be a resource identifier. The predicate $well_res$ is inductively defined by the following rules:

```

(WRlock)  well_res(r) (Wlock)  well_res(r)
(WRclaim) well_res(r) (Wclaim) well_res(r)
(WRlock)  well_res(r) (Wlock)  well_res(r)

```

The well-formedness condition for RTFMspec programs intuitively states that a Lock r statement must be followed by a well-formed critical section, followed by an Unlock r statement. Our definition of well-formedness can be seen as a formalisation of the SRP requirements as stated in [5]. The well-formedness serves as the correctness condition for the compilation as follows:

```

(Clock) comp(r) (Cunlock) comp(r)
(Cclaim) comp(r) (Cclaim) comp(r)

```

We have implemented a compilation algorithm in Why3 that is verified correct to comp.

Correct compilation of Resource Ceilings
 Tasks are captured in WhyML by the type `task_t` presented in Listing 3. The type chosen to capture the static priorities, `prio_t`, can be any type as long as it has an associated strict order. The type `task_set_t` denotes a finite set of tasks. The type `rc_t` denotes a mapping from resources to ceiling values. Finally, the type `rc_s_t` denotes the finite set of resources. Resource information are captured by the type `res_t`.

With these definitions at hand we can define a set of predicates that are necessary to address the correctness criteria for building the resource ceilings for the given set of tasks of the program at hand. These conditions are: (1) resource claims are present in the code of the program if they exist in the assumed resource set; (2) given a local minimum ceiling m , the ceiling of each of the resources considered must be at least that m ; (3) if all the tasks in the task set satisfy property (1), then property (2) must also hold; (4) finally, for each resource r with an associated ceiling c , there exists a task t with priority p claiming r . Together, these four properties serve as a formal description of ceilings according to [5].

```

type prog_t = #> #<#>
type task_t = (#> prio_t) (#> task_set_t) (#> prog_t)
type task_set_t = list task_t
type rc_t = map real_t prio_t
type rc_s_t = set real_t

```

(*) Relation over lock statements and set of resources
 $predicates\ pred_prog_rs\ (r:\ prog_t)\ (rs:\ rc_t) =$
 $forall\ r.\ mem\ r\ rs\ <>>\ Mem.\ mem\ (Lock\ r)\ p$

(*) Relation over a set of (locked) resources and their ceilings
 $predicates\ pred_prio_rc\ (n:\ prio_t)\ (rs:\ rc_t)\ (rc:\ rc_t) =$
 $forall\ r.\ mem\ r\ rs\ <>>\ r.\ rc\ r\ >= rc$

(*) Relation over the task set and resource ceilings
 $predicates\ pred_ta_rc\ (ta:\ task_set_t)\ (rc:\ rc_t) =$
 $forall\ t.\ Mem.\ mem\ t\ ta\ <>>$
 $forall\ rs.\ pred_prog_rs\ t.\ prog_rs\ <>>$
 $pred_prio_rc\ rs\ rc$

(*) Relation over resource ceilings and task priorities
 $predicates\ pred_prio_rc_prio\ (ta:\ task_set_t)\ (rc:\ rc_t) =$
 $forall\ t.\ Mem.\ mem\ t\ ta\ <>>$
 $Mem.\ mem\ t\ ta\ />\ pred_prog_rs\ t.\ prog_rs\ />\ mem\ r\ />\ rc\ (r) = t.\ prio$

Listing 3 depicts the correctness criteria. We have developed an implementation in Why3 verified to the specification.

Conclusions
 From the verified development, certified OCaml code has been extracted. Based on the extracted code an executable proof checker tool was developed. Ongoing work includes extending the RTFM-core language and certified compiler with an imperative layer (subset of C).

References

1. P. Lindgren, M. Lindner, D. Pereira, and L. M. Pinho, “RTFM-core: Language and Implementation,” in 20th IEEE Conference on Industrial Electronics and Applications (ICIEA 2015), 2015.

2. J. Erikson, F. Haggstrom, S. Arntsson, A. Krognyk, and P. Lindgren, “Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives,” in 2015 IEEE 2015, pp. 1109-1113.

3. T. Baker, “A stack-based resource allocation policy for real-time processes,” in Real Time System Symposium, 2000 Proceedings, 21st Dec. 1990, pp. 191-200.

4. M. Lindner, M. Lindner, and P. Lindgren, “RTFM-RTT: A thread-oriented approach for RTFM-core semantics execution of IEC 61499,” in IEEE International Conference on Emerging Technologies and Factory Automation, 2009 Proceedings, 2009, pp. 1-5.

5. R. W. Floyd, “Assigning meanings to programs,” Proc. Symp. Appl. Math., vol. 19, pp. 19-31, 1967.

6. C. A. R. Hoare, “An axiomatic basis for computer programming,” Commun. ACM, vol. 12, no. 10, pp. 576-580, Oct. 1969. [Online]. Available: <http://doi.acm.org/10.1145/362233.362259>



- Subset of the IEC 61499 Structured Text
- variables, expressions
- statements
 - assignments
 - conditionals
 - loops
- proven Byte code generation
- proven VM (stack machine) for Byte code
- generates MIPS assembler
 - optimal register assignment
 - proofs on the way

```
// Example  
// 1+2+3..6 = 21
```

```
WHILE a <= 5 DO  
    a := a + 1;  
    sum := sum + a  
DONE
```

- Extending front-end to larger subset of Structured Text
- Extending backend to ARM assembly
- Adopting the RTFM-4-FUN to generate RTFM-core-imp
 - ***May be proven as well, (formalising semantics of ST)***
- Integration to 4DIAC
 - Certified toolchain, provably correct implementation

Huge effort. Is it worth it?

- CompCert (C compiler) > 10 years in the making
- Commercial licence available
- We can already use CompCert with RTFM-4FUN
- Formalised semantics of C is difficult, “core-imp” simpler

Thank you for your attention!

Questions or comments

