Carnegie Mellon University
Software Engineering Institute

Mechanization of the Ravenscar Profile in Coq

JUNE 15, 2023

Jérôme Hugues

Motivation (1)

The Ada Ravenscar profile [1] is a subset of Ada concurrency to implement deterministic real-time systems:

- mono-core systems
- periodic or sporadic tasks
- communication through protected objects with Immediate Ceiling Priority Protocol, ICPP
- Absolute delays

[1] B. Dobbing, A. Burns, and T. Vardanega, “Guide for the use of the of the Ravenscar Profile in High Integrity Systems,” tech. rep., 2003.

Motivation (2)

Semantics of Ravenscar is known to be deterministic, but only ”paper” proofs, e.g. [2]: absence of deadlock, determinism, connection to Response Time Analysis, etc. At the same time, qualified Ada runtime exists.

$\Rightarrow$ How to reconcile both?

Let’s mechanize Ravenscar in a Interactive Theorem Prover

$\Rightarrow$ Reason on complex real-time systems and perform proofs, e.g., connections to other models: AADL, scheduling, etc.

[2] I. Hamid and E. Najm, “Operational semantics of Ada Ravenscar,” in Reliable Software Technologies – Ada- Europe 2008 (F. Kordon and T. Vardanega, eds.), (Berlin, Heidelberg), pp. 44–58, Springer Berlin Heidelberg, 2008.

Contributions: Mechanization of Ravenscar profile in Coq

Carnegie Mellon University Software Engineering Institute

Approach

- Extend IMP, a kernel of an imperative language with Ravenscar constructs,
- Encode semantics in rewriting and natural styles, simulation for Ravenscar functions: run-to-completion and continuation-style passing.
- Define a Ravenscar program as a collection of tasks executed by a FIFO-per-Priority simulator

Results: proof continuity from abstract semantics to continuation-passing semantics

- Size: specifications: 382 slocs | proofs: 1344 slocs
- Available as part of Oqarina: https://github.com/Oqarina/oqarina

About IMP

Toy language, mimics a (very basic) imperative language:

- Integers, Booleans, assignments, sequences, if/then/else, while
- Enough to capture key programming elements for Ravenscar
  - Integers: priority + comparison, Booleans: guards, …
  - Not full Ada, no need for more sophistication at this stage (WiP!)

$$
\begin{array}{rcl}
\text{AExp} & ::= & \overline{n} \mid x \mid (a_0 \oplus a_1) \\
\text{BExp} & ::= & \textbf{TRUE} \mid \textbf{FALSE} \mid (a_0 \odot a_1) \mid (b_0 \oslash b_1) \mid (\neg b) \\
\text{Com} & ::= & \textbf{SKIP} \mid x := a \mid (c_0 \; ; \; c_1) \mid \\
 & & (\textbf{IF } b \textbf{ THEN } c_1 \textbf{ ELSE } c_2) \mid (\textbf{WHILE } b \textbf{ DO } c \;) \\
\oplus & ::= & + \mid * \mid - \mid / \\
\odot & ::= & \leq \mid = \\
\oslash & ::= & \vee \mid \wedge
\end{array}
$$

Assignments: $$\frac{< a, \sigma > \rightarrow_a n}{< x := a, \sigma > \rightarrow < \textbf{SKIP}, \sigma[n/x] >}$$

Sequences: $$\frac{< c_0, \sigma > \rightarrow < c_0', \sigma' >}{< c_0; c_1, \sigma > \rightarrow < c_0'; c_1, \sigma' >} \qquad \frac{}{< \textbf{SKIP}; c_1, \sigma > \rightarrow < c_1, \sigma >}$$

Conditionals: $$\frac{< b, \sigma > \rightarrow_b \mathit{true}}{< \textbf{IF } b \textbf{ THEN } c_0 \textbf{ ELSE } c_1, \sigma > \rightarrow < c_0, \sigma >} \qquad \frac{< b, \sigma > \rightarrow_b \mathit{false}}{< \textbf{IF } b \textbf{ THEN } c_0 \textbf{ ELSE } c_1, \sigma > \rightarrow < c_1, \sigma >}$$

While statements: $$\frac{}{< \textbf{WHILE } b \textbf{ DO } c, \sigma > \rightarrow < \textbf{IF } b \textbf{ THEN } (c; \textbf{WHILE } b \textbf{ DO } c) \textbf{ ELSE SKIP}, \sigma >}$$

From IMP to IMP-Ravenscar

Add Ada-like notion of protected objects and delay until

COM' ::= **DELAY_UNTIL** | **PO_ENTRY) (e)**
| **PO_FUNCTION) (f)** | COM

Delay: $$\frac{}{< \textbf{DELAY\_UNTIL}, \mathcal{T} > \rightarrow < \textbf{SKIP}, \mathcal{T}[nd ::= nd + period, state := IDLE] >}$$

PO_Function: $$\frac{}{< \textbf{PO\_FUNCTION (f)}, \mathcal{T}, \mathcal{P} > \rightarrow < f, \mathcal{T}[priority ::= \mathcal{P}[priority]] >} \; \mathcal{P}[guard] \rightarrow_b true$$

PO_Entry: $$\frac{}{< \textbf{PO\_ENTRY (f)}, \mathcal{T}, \mathcal{P} > \rightarrow < f, \mathcal{T}[priority ::= \mathcal{P}[priority]] >} \; \mathcal{P}[guard] \rightarrow_b false$$
$$\frac{}{< \textbf{PO\_ENTRY (f)}, \mathcal{T}, \mathcal{P} > \rightarrow < f, \mathcal{T}[state := IDLE] >}$$

Encoding in Coq: statements

Leverage Coq type systems and constructs to define the syntax of programs (statements) as inductive types

```
Inductive statements : Type :=
  (* A sequential execution step *)
  | COMP  (WCET : Time)

  (* Operations on protected objects *)
  | PO_FUNCTION (po : identifier) (op : identifier)
  | PO_ENTRY (po : identifier) (op : identifier)

  (* Delay until some time *)
  | DELAY_UNTIL_NEXT_PERIOD

  (* Sequence of statements *)
  | SEQ (s1: statements) (s2: statements)

  (* While b is true, execute s1 *)
  | WHILE (b: bexp) (s: statements)

  | SKIP.
```

```
Example A_Program := COMP 1 ;; SKIP.

Example Ravenscar_Cyclic_Program := WHILE TRUE (COMP 2 ;;
                                                DELAY_UNTIL_NEXT_PERIOD).
```

Semantics: from the abstract to the concrete

Carnegie Mellon University Software Engineering Institute

Rewriting semantics (as a relation): match abstract semantics

```
Inductive red : statements * thread_state
                -> statements * thread_state -> Prop
```

Natural semantics : step in the execution

```
Inductive rexec: thread_state -> statements -> thread_state -> Prop
```

Evaluation #1: fuel: # iter., returns state and termination status

```
Fixpoint Eval (fuel : nat) (st :thread_state) (s : statements)
  : thread_state * bool
```

Evaluation #2: (continuation ::= ”position in program”)

```
Fixpoint Eval_cont' (fuel : nat) (st :thread_state) (k : cont)
  : thread_state * cont
```

Main proofs

Key result: all 4 semantics compute the same next/final state

- proof by induction over the statements
- #slocs could be reduced by using tactics, from 1.3K to (probably) 8'00.

But why 4 semantics?

- `red` is the "more formal"/ textbook-like, useful to reason on programs
- Ravenscar programs do not have final state (!), so `red/rexec/eval` are of limited practical interest outside of stepwise proofs, cannot derive a simulator (!)
- `Eval_cont` is the "more practical": builds on the concept of continuation
  - continuation ⇔ execution state as explicit argument, allows for "breakpoints", required for interrupting a function at a specific instruction

    ⇒ allows for "context-switching"

Simulation of Ravenscar programs

From task instance (“running” threads) to simulation (snapshot)

```
Inductive task_instance :=
| task_i: identifier -> priority -> thread_state -> cont -> task_instance.
```

```
Record ravenscar_system := {
  clock : Time ;
  taskset : list task_instance ;
}.
```

```
Definition Simulate_Ravenscar_Monocore (s : ravenscar_system) (fuel : nat) :=
  let l := taskset s in
  let elected_thread := get_thread_by_priority l in

  if (negb (task_instance_beq Idle_task_Instance elected_thread)) then
    let executed_thread := Simulate_Task elected_thread fuel in
    let time_advance := th_cet (get_thread_state executed_thread) in
    let executed_thread' := reset_cet executed_thread in {|
      clock := (clock s) + time_advance;
      taskset := replace_task_instance executed_thread' l;
    |}
```

Conclusion and future work

Simulation of mono-core systems “happens to work well”

More proofs to come:

- Encode key properties of FIFO-per-priority and ICPP, prove the simulator respects them
- Proof of correctness of ICPP implementation (bound on priority inversion)
- Connections to AADL semantics (Oqarina) and Scheduling analysis (PROSA)

(maybe) see you for AEiC 2024 for the full paper !