From RoboSim to another CSP

CLEARSY Safety Platform

Thierry Lecomte
CLEARSY
France

Marcel Oliveira
IMD/UFRN
Brazil

 thierry.lecomte@clearsy.com

 marcel@dimap.ufrn.br
Rationale

- RoboSim: diagrammatic language to model simulations of robotic systems and to generate code for use with robotic simulators
- CLEARSY Safety Platform: safety execution platform programmed with B
- On-going research[^1]:
  - Find a translation schema from RoboSim to B
  - Experiment with examples on real hardware

[^1]: *The CLEARSY Safety Platform: 5 Years of Research, Development and Deployment*
Thierry Lecomte, David Deharbe, Paulin Fournier, Marcel Oliveira
Agenda

► Introduction to the CLEARSy Safety Platform
  ▶ Principles
  ▶ Example of DSL translation
► Translation from RoboSim
  ▶ Principles
  ▶ Examples
► Perspectives
Introduction to the CLEARSY Safety Platform

Principles

► Computer with safety properties
  ▶ Used for applications where human life is at risk

► Generic architecture
  ▶ enough to be reused

► Programmed with B
  ▶ Allow « less expert engineers » to develop
  ▶ Reduced time to certification (vs standards)
History

**R&D**

CLEARSY Safety Platform building blocks certified
- 2017: platform screen door control system Sao Paulo, SIL4, CERTIFER
- 2017: platform screen door control system Stockholm, SIL3, Bureau Veritas
- 2019: vital remote I/O system, SIL4, Bureau Veritas

Development

**R&D**

CLEARSY Safety platform for Industry
- 2021: Core safety computer certified SIL4, CERTIFER
- 2022: Cross River Rail metro, Brisbane, objective SIL4

**LCHIP (Low Cost High Integrity Platform)**
Collaborative Project with SNCF
Invention of CLEARSY Safety platform for Education

**Tutorials**
Brazil, Canada, France, Italy, Norway, Portugal, UK

**Courses**
CLEARSY Safety Platform for Education released
France, Italy
Board software simulator in 2021
Safety systems ⇒ handling of failures

Many possibilities to make mistakes

- Natural language requirement
- Specification
- Design
- Source code
- Binary code

Wrong specification
Wrong program
Wrong binary
Wrong execution
Wrong environment specification
Wrong exploitation procedure

Bad hardware
Failing hardware (µC, I/O)

Many possibilities to make mistakes
Sources of failure

Using B prevents the wrong program failure

Wrong specification
Wrong program
Wrong binary
Wrong execution

Natural language requirement
Specification
Design
Source code
Binary code

Wrong environment specification
Bad hardware
Failing hardware (µC, I/O)
Wrong exploitation procedure
Failing System

Using the CLEARSY Safety Platform handles the wrong program, binary, and execution failures

- Natural language requirement
- Specification
- Design
- Source code
- Binary code
- Bad hardware
- Failing hardware (µC, I/O)

Wrong specification
Wrong program
Wrong binary
Wrong execution
Wrong environment specification
Wrong exploitation procedure
What is a safety computer?

**Strong constraints**
- SIL4: 1 catastrophic failure max per $10^4$ years
- Redundancy: 2oo2, 2oo3
- Diversity
- Responsibility

$safety \neq security$
Composition of the final binary
- The execution of the toolchain builds two independent binaries called replica.

Binary diversification
- Each replica is built by an independent and diverse compiler from the same formal model.

Runtime verification
- Both replicas are executed in sequence with the same input data. The comparison of the output data of each replica detects discrepancies and random failures during runtime.
- Cross-checking mechanisms between CPU1 and CPU2 mitigate remaining failure modes.
Architecture not new

Innovation

- Assembly, B, C intricated for both precompiled libraries and application, balanced proof
- **The software is written only once.** No need for two independent software design teams.
- Safety is out of reach of the developer who cannot alter it

Implementation in B0

- B project
- Handwritten
- translated

CPU1
- R1
- R2

CPU2
- R1
- R2

Replica 1

Replica 2

B0 compiler

C compiler

C translator

Final binary

Cross-check
Software Architecture

Main loop

- Execute F once
- Check sanity
- Check µC
- Check other µC

Data acquisition

- Processed by interrupts

Outputs control

- No underlying operating system
- Main loop is executed as fast as possible
- Binary made of
  - Replicated code: subject to 2oo2 verification
  - Non-replicated code
Software Architecture

Execute F once

Check sanity

Main loop

Check µC

Check other µC

Data acquisition

Processed by interrupts

Outputs control

CLEARSY Safety Platform for Education

- F as a B model only (replicated code)
- Most verifications implemented
- Cost effective hardware interface
- Cannot be used for real life safety app

Legend

to develop

to complement
developed or generated

CLEARSY Safety Platform

SK0
IDE + board
3 inputs, 2 outputs

SK1
IDE + board
20 inputs, 8 outputs

Robostar group I From RoboSim to another CSP
Software Architecture

Main loop

Execute F once

Check sanity

Check μC

Check other μC

Data acquisition

Outputs control

processed by interrupts

CLEARSY Safety Platform for Industry

- B and C used for sequential and interrupted code
- More adaptable to specific needs
- All required verifications implemented
- I/O hosted on a motherboard to develop

Legend
- to develop
- to complement
- developed or generated

Legend
- compilation toolchain + core computer + SK motherboard
  32 inputs, 32 outputs

Robostar group | From RoboSim to another CSP
Programming Model

- The execution is cyclic
- The function is executed regularly as often as possible similar to Arduino programming (setup(), loop())
- No underlying operating system
- No predefined cycle time (if no comms in 50ms, board enters panic mode)
- No delay()
- Inputs are values captured at the beginning of a cycle (digital I/O)
- Outputs are maintained from one cycle to another (digital I/O)

- Project skeleton is generated from board description (I/O used, naming)
- Programming is specifying and implementing the function `user_logic`

```c
init();
while (1) {
  instance1();
  instance2();
}
```
CLEARSY Safety Platform Project

The interface with the safety library

The model of the function F

Components to modify

specification

implementation

decomposition

link
Generated Models

Syntax:
\[
\text{pp} \leftarrow \text{ff}(\text{vv})
\]
represents a call to
operation \( \text{ff}(\text{vv}) \)
that returns the value \( \text{pp} \)
Model extraction from relay-based schematics
Translation from Robosim: principles

- Pragmatic approach based on examples [2]
  
  [2] *Verified Simulation for Robotics*
  Ana Cavalcanti, Augusto Sampaio, Alvaro Miyazawa, Pedro Ribeiro, Madiel Conserva Filho, André Didier, Wei Li, Jon Timmis

- Two different notions of cycles:
  - Board cycle: always running, 50 MIPS
  - Model cycle: either executing or waiting for the next model cycle
Translation from RoboSim:

1. Read all inputs and store their values in reserved input variables.
2. Write to all outputs using the values stored in reserved output variables.
3. Is this the first time the user_logic is executed?
   - YES: Reset all outputs.
   - NO: Has the model cycle timer reached the cycle duration?
     - YES: Execute the next cycle of the state machine.
     - NO: Reset the model cycle timer.
4. If NO, repeat from step 3.
User Logic

```c
user_logic =
BEGIN
    IF first_time = TRUE THEN
        reset_outputs;
        state_machine;
        cycle_timer <-- get_ms_tick;
        first_time := FALSE
    ELSE
        VAR time_elapsed, cycle_duration IN
        time_elapsed := since(cycle_timer);
        cycle_duration := mul_uint32(SimSMovement_cycle_def, cycle_unit);

        IF (cycle_duration <= time_elapsed) THEN
            reset_outputs;
            state_machine;
            cycle_timer <-- get_ms_tick
        END
    END
END;
```
The Controller State Machine

► Unless marked with the special marker event exec, RoboSim state transitions are timeless.

► This would not be respected if we simply translate the RoboSim models using a standard translation because it imposes a wait of at least one model cycle between state transitions.

► Our solution is to normalize the states with respect to the model cycles.
The Controller State Machine

Normalized State Machine
The Controller State Machine

```
state_machine =
BEGIN
  IF smstate = INIT THEN
    move(1v,0);
    smstate := EXEC_1
  ELSIF smstate = EXEC_1 THEN
    VAR local_obstacle IN
    local_obstacle := (local_obstacle:uint8_t);
    IF local_obstacle == IO_ON THEN
      MBC := get_ms_tick;
      stop;
      move(0,av);
    END
    smstate := EXEC_2
  ELSIF local_obstacle == IO_OFF THEN
    smstate := EXEC_1
  ELSE skip
  END
END
```
The Controller State Machine

```plaintext
state_machine =
BEGIN
IF smstate = INIT THEN
  move(lv,0);
  smstate := EXEC_1
ELSIF smstate = EXEC_1 THEN
  VAR local_obstacle IN
  local_obstacle:(local_obstacle:uint8_t);
  local_obstacle <- get_i_ObstacleI_obstacle;
  IF local_obstacle = IO_ON THEN
    MBC <- get_ms_tick;
    stop;
    move(0,av);
    smstate := EXEC_2
  ELSIF local_obstacle = IO_OFF THEN
    smstate := EXEC_1
  ELSE skip
END
```
The Controller State Machine

```
state_machine =
BEGIN
    IF smstate = INIT THEN
        move(1v,0);
        smstate := EXEC_1
    ELSIF smstate = EXEC_1 THEN
        VAR local_obstacle IN
            local_obstacle:(local_obstacle:uint8_t);
            local_obstacle := get_i_ObstacleI_obstacle;
            IF local_obstacle = IO_ON THEN
                MBC := get_ms_tick;
                stop;
                move(0,av);
            END
            smstate := EXEC_2
        ELSIF local_obstacle = IO_OFF THEN
            smstate := EXEC_1
        ELSE skip
    END
END
```

Robostar group | From RoboSim to another CSP
ELSIF smstate = EXEC_2 THEN
VAR since_value IN
since_value:(since_value:uint32_t);

since_value <- since(MBC);
VAR pi_div_av IN
pi_div_av:(pi_div_av:uint32_t);
pi_div_av := pi / av;
IF since_value < pi_div_av THEN
smstate := EXEC_2
ELSIF pi_div_av <= since_value THEN
move(xv,0);
smstate := EXEC_1
ELSE skip
END
END;

The Controller State Machine
The Controller State Machine

ELSIF smstate = EXEC_2 THEN
VAR since_value IN
    since_value:(since_value:uint32_t);

    since_value <= since(MBC);
VAR pi_div_av IN
    pi_div_av:(pi_div_av:uint32_t);
    pi_div_av := pi / av;
IF since_value < pi_div_av THEN
    smstate := EXEC_2
ELSIF pi_div_av <= since_value THEN
    move(1v,0);
    smstate := EXEC_1
ELSE skip
END
END skip
END;

Robostar group I From RoboSim to another CSP
Fine tuning the model cycle (obstacle avoidance)

- Essential to make inputs noticeable by the controller and to make outputs noticeable to the robotic platform.
- A long model cycle degrades the time between readings of the obstacle sensor and a short model cycle can make it impossible for the car engine to react to the command.
  - cycle_unit
  - simulation cycle definition
- Fine tuning is also necessary in the definition of the values of each of the model constants
  - namely lv, av, and pi
Fine tuning the model cycle

MACHINE user_ctx
SEES g_types
SETS STATE = {INIT, EXEC_1, EXEC_2}
CONCRETE_CONSTANTS
  // Translation constants
cycle_unit, // ms
  // Model constants
SimSMovement_cycle_def, av, lv, pi

PROPERTIES
  // Translation constants
cycle_unit: uint32_t &
  // Model constants
SimSMovement_cycle_def: uint32_t &
av: uint32_t & lv: uint32_t &
pi: uint32_t &
av: 1..7 & lv: 1..7
END

IMPLEMENTATION user_ctx_i
REFINES user_ctx
  // pragma CONSTANTS
SEES g_types
VALUES
  // Translation constants
cycle_unit = 100; // ms
  // Model Constants
SimSMovement_cycle_def = 1;
av = 7;
lv = 7;
pi = 7000
END
Operation Calls

► They are directly translated to the invocation of operations of the B implementation.
► RoboSim forbids the same operation to be invoked twice in the same cycle and our implementation resets all outputs at the beginning of every cycle.
► For each operation of the RoboSim model:
  ▶ A boolean output value indicates that it has been invoked
  ▶ The operation output values.
Encoding IOs

```
o_MovementI_move_av  o_MovementI_move_lv
```

```
o_MovementI_stop   [2]  [1]  [0]  [2]  [1]  [0]  o_MovementI_move
```

```
8  7  6  5  4  3  2  1
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

```
i_ObstacleI_obstacle
```
On going: square trajectory
Next steps: complex case studies

Square trajectory
IO, model partition

Transporter
nested state machines

Alpha algorithm
nested state machines
Perspectives

Preliminary work have demonstrated feasibility
- Principles applied to several models
- Manual translation at the moment
- POC with simple mobile robot

Further theoretical / practical issues
- Correctness of normalization (specification and correctness proof)
- Correctness of translation (CSP semantics and FDR4 verification)
- Tool support
- Impact on HW/SW interfaces (INTEGER / REAL vs Boolean)
Perspectives

▶ Rationale for industrial technology transfer
  ▶ Aimed at both functional (navigation/operation) and safety (safeguard) automatisms
  ▶ Interest from autonomous shuttle / (remotely operated)(autonomous) robots
  ▶ RoboSim as a modelling framework for exploration phase

| Education for theoretical tracks, introduction to FM | ✓ | ✓ |
| Education for embedded systems, IoT | ✓ | ✓ |
| Demonstration, PoC | ✓ | ✓ |
| Complete projects with electronics | ✓ | ✓ |
Useful References

► Github:
https://github.com/CLEARSY/tutorial-ABZ-2021

► Youtube channel:
https://www.youtube.com/channel/UCWoU4LVYy7Q7OYRp4D9FnOQ

► Web pages:
Product booklet, presentation of the CLEARSY Safety Platform, Quick Start Guide, Datasheet - vital computer board
Project RoboTIC® – Information and Communication Technology for Robotics and Applications

► Software simulator
Specialized Atelier B, board graphical animation
Thank you
For your attention