Dériver une implémentation d'une spécification

...mais pas n'importe comment...

Le 14 novembre 2024

Par Ghilain BERGERON (Université de Lorraine, CNRS, Inria, Loria)

Un peu de motivation...

Nous sommes entourés de systèmes distribués

Un peu de motivation...

Nous sommes entourés de systèmes distribués *contenant des bugs.*

Un peu de motivation...

* Intérêt grandissant dans la vérification de ces systèmes (p.e. avec TLA$^+$ ou Event-B).

Un peu de motivation...

* Intérêt grandissant dans la vérification de ces systèmes (p.e. avec TLA$^+$ ou Event-B).
* Le problème :

Spécification (en TLA$^+$)

Implémentation (en Go)

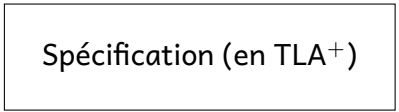
Un peu de motivation...

* Intérêt grandissant dans la vérification de ces systèmes (p.e. avec TLA$^+$ ou Event-B).
* Le problème :

Un peu de motivation...

* Intérêt grandissant dans la vérification de ces systèmes (p.e. avec TLA$^+$ ou Event-B).
* Le problème :

* Une solution :



Un peu de motivation...

* Intérêt grandissant dans la vérification de ces systèmes (p.e. avec TLA$^+$ ou Event-B).
* Le problème :

* Une solution :

Compilation vérifiée

* Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$

Compilation vérifiée

* Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées
  $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et
  $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.

Compilation vérifiée

* Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées
  $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et
  $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
* Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.

Compilation vérifiée

* Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées
  $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et
  $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
* Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
* Soit $\mathcal{C} : \mathcal{L}_1 \rightharpoonup \mathcal{L}_2$ une fonction (partielle) de compilation.

Compilation vérifiée

- Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées
  $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et
  $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{C} : \mathcal{L}_1 \rightharpoonup \mathcal{L}_2$ une fonction (partielle) de compilation.

$\mathcal{C}$ préserve la sémantique (selon $\mathcal{R}$) si

Compilation vérifiée

- Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées
  $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et
  $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{C} : \mathcal{L}_1 \rightharpoonup \mathcal{L}_2$ une fonction (partielle) de compilation.

$\mathcal{C}$ préserve la sémantique (selon $\mathcal{R}$) si

- Pour tout programme $S \in \mathcal{L}_1$, ...

Compilation vérifiée

- Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées
  $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et
  $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{C} : \mathcal{L}_1 \rightharpoonup \mathcal{L}_2$ une fonction (partielle) de compilation.

$\mathcal{C}$ préserve la sémantique (selon $\mathcal{R}$) si

- Pour tout programme $S \in \mathcal{L}_1$, ...
- ...si $\mathcal{C}$ est définie sur S...

Compilation vérifiée

- Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées
  $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et
  $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{C} : \mathcal{L}_1 \rightharpoonup \mathcal{L}_2$ une fonction (partielle) de compilation.

$\mathcal{C}$ préserve la sémantique (selon $\mathcal{R}$) si

- Pour tout programme $S \in \mathcal{L}_1$, ...
- ...si $\mathcal{C}$ est définie sur S...
- ...alors pour tout couple $(\sigma_2, \varepsilon_2, r_2) \in [\![\mathcal{C}(S)]\!]$, ...

Compilation vérifiée

- Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{C} : \mathcal{L}_1 \rightharpoonup \mathcal{L}_2$ une fonction (partielle) de compilation.

$\mathcal{C}$ préserve la sémantique (selon $\mathcal{R}$) si

- Pour tout programme $S \in \mathcal{L}_1$, ...
- ...si $\mathcal{C}$ est définie sur S...
- ...alors pour tout couple $(\sigma_2, \varepsilon_2, r_2) \in [\![\mathcal{C}(S)]\!]$, ...
- ...il existe un couple $(\sigma_1, \varepsilon_1, r_1) \in [\![S]\!]$ tel que $((\sigma_1, \varepsilon_1, r_1), (\sigma_2, \varepsilon_2, r_2)) \in \mathcal{R}$.

Compilation vérifiée

- Soient deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ et leurs sémantiques associées $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{P}(\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1)$ et $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{P}(\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{R} : (\mathit{State}_1 \times \mathit{Trace} \times \mathit{Result}_1) \times (\mathit{State}_2 \times \mathit{Trace} \times \mathit{Result}_2)$.
- Soit $\mathcal{C} : \mathcal{L}_1 \rightharpoonup \mathcal{L}_2$ une fonction (partielle) de compilation.

$\mathcal{C}$ préserve la sémantique (selon $\mathcal{R}$) si

- Pour tout programme $S \in \mathcal{L}_1$, ...
- ...si $\mathcal{C}$ est définie sur S...
- ...alors pour tout couple $(\sigma_2, \varepsilon_2, r_2) \in [\![\mathcal{C}(S)]\!]$, ...
- ...il existe un couple $(\sigma_1, \varepsilon_1, r_1) \in [\![S]\!]$ tel que $((\sigma_1, \varepsilon_1, r_1), (\sigma_2, \varepsilon_2, r_2)) \in \mathcal{R}$.

$$\forall S \in \mathcal{L}_1.\ [\![\mathcal{C}(S)]\!] \subseteq \mathcal{R}[[\![S]\!]]$$

* Front-end à TLA^+.

* Front-end à TLA^+.
* Facilite l'écriture de machines à états, avec un DSL proche de C/Pascal.

* Front-end à TLA$^+$.
* Facilite l'écriture de machines à états, avec un DSL proche de C/Pascal.

Court exemple : Producer-Consumer

```
(*--algorithm ProducerConsumer {
    fifo
        \* @type Channel[Nat]
        Consumer;

    process (P = 0)
        variable
            \* @type Nat
            x = 0;
    {
p:      while (TRUE) {
            send(Consumer, x);
            x := x + 1;
        }
    }

        \* @mailbox Consumer
    process (C = 1)
        variable
            \* @type Nat
            y = 0;
    {
c:      while (TRUE) {
            receive(Consumer, y);
            print y;
        }
    }
}*)
```

* Un bloc atomique `label: instructions+` doit pouvoir s'exécuter complètement.

* Un bloc atomique `label: instructions+` doit pouvoir s'exécuter complètement.
* Un seul bloc atomique ne s'exécute à la fois.

* Un bloc atomique `label: instructions+` doit pouvoir s'exécuter complètement.
* Un seul bloc atomique ne s'exécute à la fois.

* $State = \mathcal{P}(PId \times \mathcal{P}(Label))$

* Un bloc atomique `label: instructions+` doit pouvoir s'exécuter complètement.
* Un seul bloc atomique ne s'exécute à la fois.

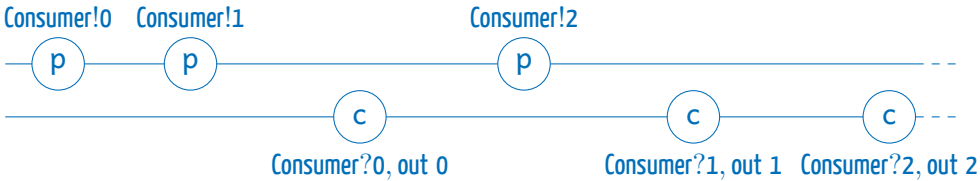
* $\mathit{State} = \mathcal{P}(\mathit{PId} \times \mathcal{P}(\mathit{Label}))$
* $\mathit{Trace} = (\_!\_ \mid \_?\_ \mid \mathsf{out}\ \_)^*$

* Un bloc atomique `label: instructions+` doit pouvoir s'exécuter complètement.
* Un seul bloc atomique ne s'exécute à la fois.

* $\mathit{State} = \mathcal{P}(\mathit{PId} \times \mathcal{P}(\mathit{Label}))$
* $\mathit{Trace} = (\_!\_ \mid \_?\_ \mid \mathsf{out}\ \_)^*$
* $\mathit{Result} = \mathbb{B}$

```
(*--algorithm ProducerConsumer {
    fifo
        \* @type Channel[Nat]
        Consumer;

    process (P = 0)
        variable
            \* @type Nat
            x = 0;
    {
p:      while (TRUE) {
            send(Consumer, x);
            x := x + 1;
        }
    }
        \* @mailbox Consumer
        process (C = 1)
            variable
                \* @type Nat
                y = 0;
        {
c:          while (TRUE) {
                receive(Consumer, y);
                print y;
            }
        }
}*)
```

```
(*--algorithm ProducerConsumer {
    fifo
        \* @type Channel[Nat]
        Consumer;

    process (P = 0)
        variable
            \* @type Nat
            x = 0;
    {
p:      while (TRUE) {
            send(Consumer, x);
            x := x + 1;
        }
    }

        \* @mailbox Consumer
        process (C = 1)
            variable
                \* @type Nat
                y = 0;
        {
c:          while (TRUE) {
                receive(Consumer, y);
                print y;
            }
        }
}*)
```



Comment compiler ce bloc ?

```
l:      x := x + 5;
        await x > 10;
        either {
            receive(C, y);
            await y # 0;
            print y;
        } or {
            y := x + 3;
        }
```

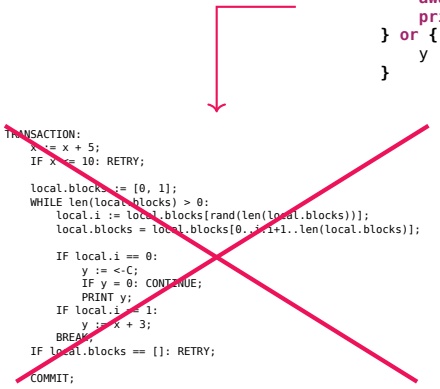
Comment compiler ce bloc ?

```
l:      x := x + 5;
        await x > 10;
        either {
            receive(C, y);
            await y # 0;
            print y;
        } or {
            y := x + 3;
        }
```

```
TRANSACTION:
    x := x + 5;
    IF x <= 10: RETRY;

    local.blocks := [0, 1];
    WHILE len(local.blocks) > 0:
        local.i := local.blocks[rand(len(local.blocks))];
        local.blocks = local.blocks[0..i:i+1..len(local.blocks)];

        IF local.i == 0:
            y := <-C;
            IF y = 0: CONTINUE;
            PRINT y;
        IF local.i == 1:
            y := x + 3;
        BREAK;
    IF local.blocks == []: RETRY;

    COMMIT;
```

Comment compiler ce bloc ?

```
l:      x := x + 5;
        await x > 10;
        either {
            receive(C, y);
            await y # 0;
            print y;
        } or {
            y := x + 3;
        }
```



```
TRANSACTION:
    x := x + 5;
    IF x <= 10: RETRY;

    local.blocks := [0, 1];
    WHILE len(local.blocks) > 0:
        local.i := local.blocks[rand(len(local.blocks))];
        local.blocks = local.blocks[0..i-1+1..len(local.blocks)];

        IF local.i == 0:
            y := <-C;
            IF y = 0: CONTINUE;
            PRINT y;
        IF local.i == 1:
            y := x + 3;
        BREAK;
    IF local.blocks == []: RETRY;

    COMMIT;
```

Guarded PlusCal

Guarded PlusCal

Distributed PlusCal mais...

Distributed PlusCal mais... tous les blocs atomiques sont de la forme

$$L : \textit{either } \{\textit{await } C_1;\ B_1;\ \textit{goto } L_1\} \textit{ or } \ldots \textit{ or } \{\textit{await } C_n;\ B_n;\ \textit{goto } L_n\}$$

Guarded PlusCal mais...

Network PlusCal

Guarded PlusCal mais… un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Guarded PlusCal mais… un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Exemple :

```
    process (C = 1)
        variable
            \* @type Nat
            y = 0;
    {
c:      receive(Consumer, y);
        print y;
        goto c;
    }
}*)
```

Guarded PlusCal mais… un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Exemple :

```
    process (C = 1)
        variable
            \* @type Nat
            y = 0;
    {
c:      receive(Consumer, y);
        print y;
        goto c;
    }
}*)
```

$\mathcal{C}_{G \to N}$

Guarded PlusCal mais… un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Exemple :

```
    process (C = 1)
        variable
            \* @type Nat
            y = 0;
    {
c:      receive(Consumer, y);
        print y;
        goto c;
    }
}*)
```

$\mathcal{C}_{G \to N}$

```
    process (C = 1)
        variables
            \* @type Nat
            y = 0,
            \* @type Seq[Nat]
            \* @inbox
            inbox = <<>>,
            \* @type Nat
            inbox_rx = 0;
    {
c:      await Len(inbox) > 0;
        y := Head(inbox);
        inbox := Tail(inbox);
        print y;
        goto c;
    }
    \* @rx
    {
rx:     receive(Consumer, inbox_rx);
        inbox := Append(inbox, inbox_rx);
        goto rx;
    }
```

Network PlusCal

Guarded PlusCal mais… un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Est-ce que $\mathcal{C}_{G \to N}$ préserve la sémantique de Guarded PlusCal ?

Guarded PlusCal mais... un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Est-ce que $\mathcal{C}_{G \to N}$ préserve la sémantique de Guarded PlusCal ?

$$\mathcal{R}_{G \to N} = \left\{ ((\sigma_G, \varepsilon_G, r_G), (\sigma_N, \varepsilon_N, r_N)) \;\middle|\; \begin{array}{c} \sigma_G = \lfloor \sigma_N \rfloor_{\text{inbox,inbox\_rx,outbox,outbox\_tx}} \\ r_G = r_N \\ \varepsilon_N \in \mathit{Sym}(\varepsilon_G) \end{array} \right\}$$

Guarded PlusCal mais... un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Est-ce que $\mathcal{C}_{G \to N}$ préserve la sémantique de Guarded PlusCal ?

$$\mathcal{R}_{G \to N} = \left\{ ((\sigma_G, \varepsilon_G, r_G), (\sigma_N, \varepsilon_N, r_N)) \;\middle|\; \begin{array}{c} \sigma_G = \lfloor \sigma_N \rfloor_{\text{inbox,inbox\_rx,outbox,outbox\_tx}} \\ r_G = r_N \\ \varepsilon_N \in \mathit{Sym}(\varepsilon_G) \end{array} \right\}$$

Guarded PlusCal mais... un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Est-ce que $\mathcal{C}_{G \to N}$ préserve la sémantique de Guarded PlusCal ?

$$\mathcal{R}_{G \to N} = \left\{ ((\sigma_G, \varepsilon_G, r_G), (\sigma_N, \varepsilon_N, r_N)) \;\middle|\; \begin{array}{c} \sigma_G = \lfloor \sigma_N \rfloor_{\text{inbox,inbox\_rx,outbox,outbox\_tx}} \\ r_G = r_N \\ \varepsilon_N \in \mathit{Sym}(\varepsilon_G) \end{array} \right\}$$

Guarded PlusCal mais... un thread supplémentaire par process, pour gérer la réception/l'envoi de messages.

Est-ce que $\mathcal{C}_{G \to N}$ préserve la sémantique de Guarded PlusCal ?

$$\mathcal{R}_{G \to N} = \left\{ ((\sigma_G, \varepsilon_G, r_G), (\sigma_N, \varepsilon_N, r_N)) \;\middle|\; \begin{array}{c} \sigma_G = \lfloor \sigma_N \rfloor_{\text{inbox,inbox\_rx,outbox,outbox\_tx}} \\ r_G = r_N \\ \varepsilon_N \in \mathit{Sym}(\varepsilon_G) \end{array} \right\}$$

Oui !

Guarded PlusCal mais...

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

Exemple :

```
c:      await Len(inbox) > 0;
        y := Head(inbox);
        inbox := Tail(inbox);
        print y;
        goto c;
```

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

Exemple :

```
c:      await Len(inbox) > 0;
        y := Head(inbox);
        inbox := Tail(inbox);
        print y;
        goto c;
```

$\mathcal{C}_{N \to Go}$

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

Exemple :

```
c:      await Len(inbox) > 0;
        y := Head(inbox);
        inbox := Tail(inbox);
        print y;
        goto c;
```

$\mathcal{C}_{N \to Go}$

```
func C_c(self : address, st : ref CState, Done : chan unit) {
    let done = make(ref bool, false);

    while !*done {
        lock st→inbox_lock;

        if !(len(st→inbox) > 0) {
            goto retry;
        }

        st→y = st→inbox[0];
        st→inbox = st→inbox[1…];
        print(st→y);
        done = true;

retry:
        unlock st→inbox_lock;
    }

    go C_c(self, st, Done);
}
```

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

```
    process (C = 1)
        variables
            \* @type Nat
            y = 0,
            \* @type Seq[Nat]
            \* @inbox
            inbox = <<>>,
            \* @type Nat
            inbox_rx = 0;
    {
c:      …
    }
    {
rx:     …
    }
```

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

```
    process (C = 1)
        variables
            \* @type Nat
            y = 0,
            \* @type Seq[Nat]
            \* @inbox
            inbox = <<>>,
            \* @type Nat
            inbox_rx = 0;
    {
c:      …
    }
    {
rx:     …
    }
```

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

```
    process (C = 1)
        variables
            \* @type Nat
            y = 0,
            \* @type Seq[Nat]
            \* @inbox
            inbox = <<>>,
            \* @type Nat
            inbox_rx = 0;
    {
c:      …
    }
    {
rx:     …
    }
```

$\mathcal{C}_{N \to Go}$

```
func C(self : address) : (ref array int, lock, chan unit) {
    let Done1 = make(chan unit, 1);
    let Done2 = make(chan unit, 2);
    let inbox = make(array int);
    let lock1 = make(lock);
    let st = make(ref CState, CState{
        y: 0,
        inbox: inbox,
        inbox_lock: lock1,
        inbox_rx: 0
    });

    go C_c(self, st, Done2);
    go C_rcv(self, st, Done2);
    go {
        let _ = ←Done2;
        let _ = ←Done2;
        Done1 ← unit{};
    };

    return inbox, lock1, Done1
}
```

Guarded PlusCal mais... introduction de coroutines et de verrous pour éviter les *data races*.

Est-ce que $\mathcal{C}_{N \to Go}$ préserve la sémantique de Network PlusCal ?

Guarded PlusCal mais... introduction de coroutines et de verrous pour éviter les *data races*.

Est-ce que $\mathcal{C}_{N \to Go}$ préserve la sémantique de Network PlusCal ?

$$\mathcal{R}_{N \to Go} = \left\{ ((\sigma_N, \varepsilon_N, r_N), (\sigma_{Go}, \varepsilon_{Go}, r_{Go})) \;\middle|\; ? \right\}$$

Guarded PlusCal mais… introduction de coroutines et de verrous pour éviter les *data races*.

Est-ce que $\mathcal{C}_{N \to Go}$ préserve la sémantique de Network PlusCal ?

$$\mathcal{R}_{N \to Go} = \left\{ ((\sigma_N, \varepsilon_N, r_N), (\sigma_{Go}, \varepsilon_{Go}, r_{Go})) \;\middle|\; ? \right\}$$

Ça devrait !

Conclusion

* 2 passes de compilation : $\mathcal{C}_{G \to N}$ et $\mathcal{C}_{N \to Go}$...

Conclusion

* 2 passes de compilation : $\mathcal{C}_{G \to N}$ et $\mathcal{C}_{N \to Go}$...
  ...chacune préservant la sémantique de leurs langages sources.

Conclusion

* 2 passes de compilation : $\mathcal{C}_{G \to N}$ et $\mathcal{C}_{N \to Go}$...

  ...chacune préservant la sémantique de leurs langages sources.

* Par composition, le compilateur $\mathcal{C}_{G \to N} \, ; \mathcal{C}_{N \to Go}$ préserve la sémantique :

Conclusion

* 2 passes de compilation : $\mathcal{C}_{G \to N}$ et $\mathcal{C}_{N \to Go}$...

  ...chacune préservant la sémantique de leurs langages sources.

* Par composition, le compilateur $\mathcal{C}_{G \to N} \,;\, \mathcal{C}_{N \to Go}$ préserve la sémantique :

  Pour tout programme $S \in$ Guarded PlusCal,
  $[\![(\mathcal{C}_{G \to N} \,;\, \mathcal{C}_{N \to Go})(S))]\!]_{Go} \subseteq (\mathcal{R}_{G \to N} \,;\, \mathcal{R}_{N \to Go})[[\![S]\!]_G]$.

Conclusion

* 2 passes de compilation : $\mathcal{C}_{G\to N}$ et $\mathcal{C}_{N\to Go}$...

  ...chacune préservant la sémantique de leurs langages sources.

* Par composition, le compilateur $\mathcal{C}_{G\to N} \,;\, \mathcal{C}_{N\to Go}$ préserve la sémantique :

  Pour tout programme $S \in$ Guarded PlusCal,
  $[\![(\mathcal{C}_{G\to N} \,;\, \mathcal{C}_{N\to Go})(S))]\!]_{Go} \subseteq (\mathcal{R}_{G\to N} \,;\, \mathcal{R}_{N\to Go})[[\![S]\!]_G]$.

$\Rightarrow$ Formalisation en Lean en cours.

Conclusion

* 2 passes de compilation : $\mathcal{C}_{G\to N}$ et $\mathcal{C}_{N\to Go}$...

  ...chacune préservant la sémantique de leurs langages sources.

* Par composition, le compilateur $\mathcal{C}_{G\to N} \,;\, \mathcal{C}_{N\to Go}$ préserve la sémantique :

  Pour tout programme $S \in$ Guarded PlusCal,
  $[\![(\mathcal{C}_{G\to N} \,;\, \mathcal{C}_{N\to Go})(S))]\!]_{Go} \subseteq (\mathcal{R}_{G\to N} \,;\, \mathcal{R}_{N\to Go})[[\![S]\!]_G]$.

⇒ Formalisation en Lean en cours.

* Peut-on étendre le langage supporté par le compilateur à Distributed PlusCal ?

Simplified PlusCal

Distributed PlusCal mais...

Distributed PlusCal mais... tous les *goto* sont explicites et les boucles *while* sont éliminées.

Distributed PlusCal mais... tous les *goto* sont explicites et les boucles *while* sont éliminées.

$$\mathcal{C}_{D\to S}(x : \textit{while } e \textit{ do } B_1;\ B_2) =$$
$$\quad x : \textit{if } e \textit{ then } \{\mathcal{C}_{D\to S}(B_1);\ \textit{goto } x\} \textit{ else } \mathcal{C}_{D\to S}(B_2)$$

Flow PlusCal

Simplified PlusCal mais...

Simplified PlusCal mais… les *if* sont éliminés et les blocs *either* sont aplatis.

Simplified PlusCal mais... les *if* sont éliminés et les blocs *either* sont aplatis.

$$\mathcal{C}_{S\to F}(\textit{if } e \textit{ then } B_1 \textit{ else } B_2) =$$
$$\quad \textit{either } \{\textit{await } e;\ \mathcal{C}_{S\to F}(B_1)\} \textit{ or } \{\textit{await } \neg e;\ \mathcal{C}_{S\to F}(B_2)\}$$

Simplified PlusCal mais... les *if* sont éliminés et les blocs *either* sont aplatis.

$$\mathcal{C}_{S\to F}(\mathit{if}\ e\ \mathit{then}\ B_1\ \mathit{else}\ B_2) = \\ \quad \mathit{either}\ \{\mathit{await}\ e;\ \mathcal{C}_{S\to F}(B_1)\}\ \mathit{or}\ \{\mathit{await}\ \neg e;\ \mathcal{C}_{S\to F}(B_2)\}$$

$$\mathcal{C}_{S\to F}(\mathit{either}\ B_1\ \mathit{or}\ \ldots\ \mathit{or}\ \{B_i;\ \mathit{either}\ B_{i_1}\ \mathit{or}\ \ldots\ \mathit{or}\ B_{i_j};\ B_i'\}\ \mathit{or}\ \ldots\ \mathit{or}\ B_n) = \\ \quad \mathit{either}\ B_1\ \mathit{or}\ \ldots\ \mathit{or}\ \{B_i;\ B_{i_1};\ B_i'\}\ \mathit{or}\ \ldots\ \mathit{or}\ \{B_i;\ B_{i_j};\ B_i'\}\ \mathit{or}\ \ldots\ \mathit{or}\ B_n$$

Simplified PlusCal mais... les *if* sont éliminés et les blocs *either* sont aplatis.

$$\mathcal{C}_{S\to F}(\mathit{if}\ e\ \mathit{then}\ B_1\ \mathit{else}\ B_2) = \\ \quad \mathit{either}\ \{\mathit{await}\ e;\ \mathcal{C}_{S\to F}(B_1)\}\ \mathit{or}\ \{\mathit{await}\ \neg e;\ \mathcal{C}_{S\to F}(B_2)\}$$

$$\mathcal{C}_{S\to F}(\mathit{either}\ B_1\ \mathit{or}\ \ldots\ \mathit{or}\ \{B_i;\ \mathit{either}\ B_{i_1}\ \mathit{or}\ \ldots\ \mathit{or}\ B_{i_j};\ B_i'\}\ \mathit{or}\ \ldots\ \mathit{or}\ B_n) = \\ \quad \mathit{either}\ B_1\ \mathit{or}\ \ldots\ \mathit{or}\ \{B_i;\ B_{i_1};\ B_i'\}\ \mathit{or}\ \ldots\ \mathit{or}\ \{B_i;\ B_{i_j};\ B_i'\}\ \mathit{or}\ \ldots\ \mathit{or}\ B_n$$

On obtient ensuite Guarded PlusCal en faisant flotter les *await*/*receive* le plus haut possible dans les branches du seul *either* restant.