
Bedwyr, a proof-search approach to model-checking

David Baelde, INRIA / École Polytechnique

Bedwyr was developed with Gacek, Miller, Nadathur & Tiu.

Bedwyr is a proof-search engine featuring :

- finite failure thanks to definitions ;
- λ -tree approach to HOAS ;
- reasoning about generic variables thanks to ∇ ;
- and tabling.

It allows to execute and reason about logic specifications of :

- λ -calculus : typing and evaluation ;
- π -calculus : transitions, (bi)simulations ;
- provability in object logics, e.g. HH ;
- model-checking on graphs, winning strategies in games, etc.

(And Bedwyr is a knight of the round table, known as a not-so-sound logician in *The Quest for the Holy Grail...*)

The logic is parametrized by a set of definitions :

$$nat \triangleq \lambda x. x = 0 \vee \exists y. x = s y \wedge nat y$$

Some unusual rules in FOLDN, nothing really new to implement :

$$\frac{\Gamma, (\sigma, x) \triangleright Hx \vdash \sigma' \triangleright G}{\Gamma, \sigma \triangleright \forall x. Hx \vdash \sigma' \triangleright G} \quad \frac{\Gamma \vdash (\sigma, x) \triangleright Gx}{\Gamma \vdash \sigma \triangleright \forall x. Gx}$$

$$\frac{\Gamma, \sigma \triangleright B\vec{t} \vdash \sigma' \triangleright G}{\Gamma, \sigma \triangleright p\vec{t} \vdash \sigma' \triangleright G} \quad \frac{\Gamma \vdash \sigma \triangleright B\vec{t}}{\Gamma \vdash \sigma \triangleright p\vec{t}} \quad p \triangleq B$$

$$\frac{\Sigma, h; \Gamma, \sigma \triangleright F(h\sigma) \vdash \sigma' \triangleright G}{\Sigma; \Gamma, \sigma \triangleright \exists x. Fx \vdash \sigma' \triangleright G} \quad \frac{\Sigma; \Gamma \vdash \sigma \triangleright G(t\sigma)}{\Sigma; \Gamma \vdash \sigma \triangleright \exists x. Gx}$$

$$\frac{\{(\Gamma \vdash \sigma' \triangleright G)\theta : \theta \in csu(\lambda\sigma. s \doteq \lambda\sigma. t)\}}{\Gamma, \sigma \triangleright s = t \vdash \sigma' \triangleright G} \quad \frac{}{\Gamma \vdash \sigma \triangleright t = t}$$

Reasoning about provability in HH

Let's define Hereditary Harrop provability in Bedwyr :

$\text{seq } L \text{ (forall } B) := \text{nabla } x \setminus \text{seq } L \text{ (} B \text{ } x) .$

$\text{seq } L \text{ (} D \text{ } \rightarrow \text{ } G) := \text{seq (and } D \text{ } L) \text{ } G .$

$\text{seq } L \text{ } A := \text{atom } A, \text{ bc } L \text{ } L \text{ } A .$

...

Not much thinking is needed to prove that

$\text{pi } t \setminus \text{pi } u \setminus \text{pi } w \setminus$

$\text{seq } tt \text{ (forall } x \setminus \text{forall } y \setminus (\text{p } x \text{ } t) \rightarrow (\text{p } y \text{ } u) \rightarrow (\text{p } x \text{ } w))$

$\Rightarrow w = t$

Unfold the definition of seq on the left, two cases remain :

$$\begin{array}{l|l} x, y \triangleright bc \Gamma pxt pxw \vdash w = t & x, y \triangleright bc \Gamma pyu pxw \vdash w = t \\ x, y \triangleright pxt = pxw \vdash w = t & x, y \triangleright pyu = pxw \vdash w = t \\ \lambda x. \lambda y. pxt = \lambda x. \lambda y. pxw & \lambda x. \lambda y. pyu = \lambda x. \lambda y. pxw \end{array}$$

Bedwyr searches for proofs in a fragment of FOLDN. Given its power, one may still call it a (pure) logic programming language.

$$\begin{aligned} \mathcal{L}0 & ::= \mathcal{L}0 \wedge \mathcal{L}0 \mid \mathcal{L}0 \vee \mathcal{L}0 \mid s = t \mid p\vec{t} \\ & \mid \nabla x.\mathcal{L}0x \mid \exists x.\mathcal{L}0x \\ \mathcal{L}1 & ::= \mathcal{L}1 \wedge \mathcal{L}1 \mid \mathcal{L}1 \vee \mathcal{L}1 \mid s = t \mid p\vec{t} \\ & \mid \nabla x.\mathcal{L}1x \mid \exists x.\mathcal{L}1x \\ & \mid \forall x.\mathcal{L}1x \mid \mathcal{L}0 \supset \mathcal{L}1 \end{aligned}$$

Implicitly : syntactic conditions on the bodies of the defined atoms p .

On the left of the implication there are only *invertible* connectives. The strategy is to introduce them eagerly.

How to find θ (ranging over logic variables) such that $\vdash (A \supset B)\theta$?

1. Collect all σ_i (ranging over eigenvariables) such that $\vdash A\sigma_i$.
2. Find θ such that for all i , $\vdash B\sigma_i\theta$.

In particular a finite failure on a level-0 formula F yields success on $F \supset \perp$.

Bedwyr's engine is actually a standard depth-first proof-search procedure, except that :

- it carries the extra generic context ;
- it only accepts \forall and \supset in right-mode ;
- it unifies logic variables on the right, eigenvariables on the left.

λ Prolog does not support case-analysis or negation-as-failure :

```
p (f a).
```

```
p (f b).
```

```
?- forall x\ p x -> exists y\ x = f y.
```

On the other hand, Bedwyr *always* does a deep case-analysis :

```
nat z.
```

```
nat (s X) := nat X.
```

```
?= pi x\ nat x => nat x.
```

Bedwyr suffers from the usual incompletenesses of depth-first engines, but also from more specific problems.

- How to handle logic variables on the left ?

$$X=1 \Rightarrow X=1$$

We would need to mix disunification and unification, there would easily be an infinity of solutions... so we just give up.

- We restrict ourselves to higher-order patterns, and give up on more complicated problems.

We use Nadathur and Linnell's implementation, which makes use of a level annotation to represent raising efficiently. We extended it with ∇ indices, local level annotations and corresponding constraints, which allows to avoid errors on goals like

$$\nabla y \ \sigma a \ \pi x \ a \ x = a \ x$$

- Finally, we must check that the instantiations of eigenvariables on the left hand-side do not make right hand-side problems fall outside of higher-order patterns...

We are currently experimenting with tabling, in order to avoid redundant and cyclic computations.

When you explicitly declare a definition to be `inductive` or `coinductive`, Bedwyr will remember the proved/disproved instances of the definition but also the encountered ones for loop detection.

$$\frac{\frac{?}{\vdash d \bar{x}}}{\vdash d \bar{x}}$$

Loops on inductive definitions are a failure, but they yield success for coinductive ones.

Tabling is the only use of the (co)induction rules of LINC.

Miller and Tiu's formalization of open bisimulation for π -calculus in LINC fits in Level 0/1. The one-step transition specification is within Level 0, and bisimulation roughly goes as follows :

```
coinductive bisim P Q :=
  (pi A \ pi P1 \ step P A P1 =>
    sigma Q1 \ step Q A Q1, bisim P1 Q1),
  (pi A \ pi Q1 \ step Q A Q1 =>
    sigma P1 \ step P A P1, bisim P1 Q1).
```

It means that writing it down in Bedwyr will give an *executable specification* of it, that is a bisimulation checker. All that without knowing any implementation detail, the ability to modify the spec easily, etc.

% bound input

onep (in X M) (dn X) M.

% free output

one (out X Y P) (up X Y) P.

% comm

one (par P Q) tau (par (M Y) T) :=
 onep P (dn X) M & one Q (up X Y) T.

% open

onep (nu x\M x) (up X) N :=
 nabla y\ one (M y) (up X y) (N y).

% close

one (par P Q) tau (nu y\ par (M y) (N y)) :=
 sigma X\ onep P (dn X) M & onep Q (up X) N.

The real specification of simulation is as follows :

$$\begin{aligned} \text{coinductive sim } P \ Q := & \ \text{pi } A \ \backslash \ \text{pi } P1 \ \backslash \ \text{pi } M \ \backslash \\ & \ (\text{one } P \ A \ P1 \Rightarrow \text{one } Q \ A \ Q1 \ \& \ \text{sim } P1 \ Q1), \\ & \ (\text{onep } P \ (\text{dn } X) \ M \Rightarrow \text{onep } Q \ (\text{dn } X) \ N, \\ & \ \text{pi } w \ \backslash \ \text{sim } (M \ w) \ (N \ w)), \\ & \ (\text{onep } P \ (\text{up } X) \ M \Rightarrow \text{onep } Q \ (\text{up } X) \ N, \\ & \ \text{nabla } w \ \backslash \ \text{sim } (M \ w) \ (N \ w)). \end{aligned}$$

Bisimulation is twice as large but similar.

Again, binding and freshness issues are completely expressed by the three binders of LINC, as shown in simple examples :

$$\begin{aligned} a(x).a(y).0 & \sim a(x).vz.a(y).0 \\ a(x).vy.[x = y].P & \sim a(x).0 \\ vx.\bar{a}(x).c(y).[x = y].P & \not\sim vx.\bar{a}(x).c(y).0 \end{aligned}$$

More complex examples involving weak bisimulation and encodings of natural numbers benefit a lot from tabling... but we still can't compete with dedicated tools such as MWB.

```
% 5 + 5 = 10
```

```
#assert
```

```
(weak_bisim
```

```
(church s z
```

```
(ss (ss (ss (ss (ss (ss (ss (ss (ss (ss (ss zz))))))))))))))
```

```
(nu s1\ nu z1\ nu s2\ nu z2\  
(par (church s1 z1 (ss (ss (ss (ss (ss zz))))))
```

```
(par (church s2 z2 (ss (ss (ss (ss (ss zz))))))
```

```
(add s1 z1 s2 z2 s z))))).
```

Regarding Bedwyr :

- ongoing work on tabling : make it sound, extend it ;
- suspend non- $L\lambda$ unifications ;
- try to generalize and re-use the term unification library ;

Beyond Bedwyr :

- work on the restrictions of LINC's (co)induction, or move to LG ;
- design tools with real support for (co)induction, but still using focused proof-search disciplines.

Thank you !