Feature-based modular verification of software product lines

Maurice ter Beek

ISTI–CNR, Pisa

3rd General Meeting CINA, Torino

February 12, 2015

joint work with Erik de Vink and Tessa Belder
(TU Eindhoven & CWI Amsterdam, The Netherlands)

SPLE: Software Product Line Engineering

(Software) Product Line Engineering

develop a product line (a.k.a. a family) using a shared architecture or platform (commonalities) and mass customization (variabilities) to serve, e.g., different markets, thus allowing for (software) reuse

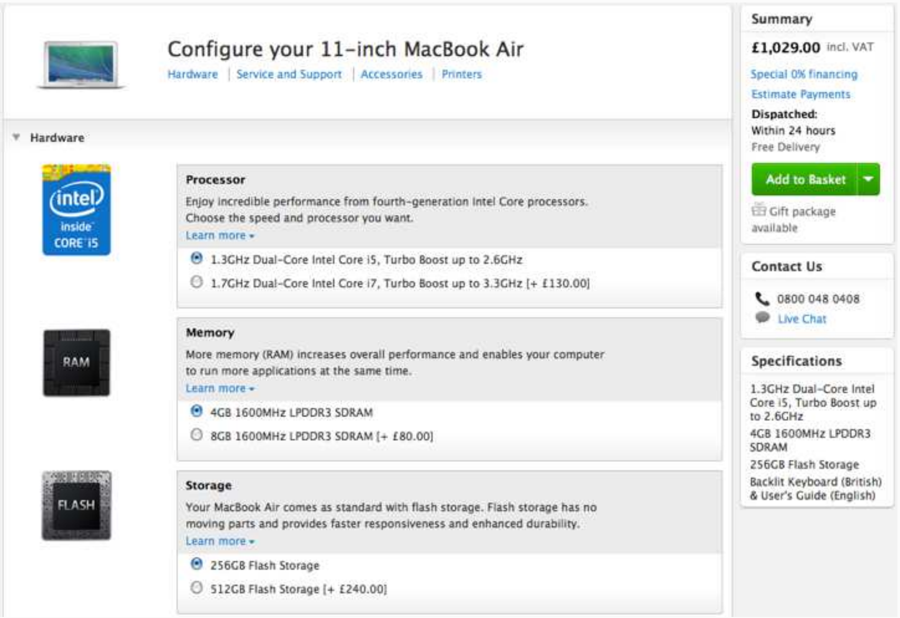
aim: maximize commonalities whilst minimizing cost of variations (i.e. of individual products)

variability in terms of features:

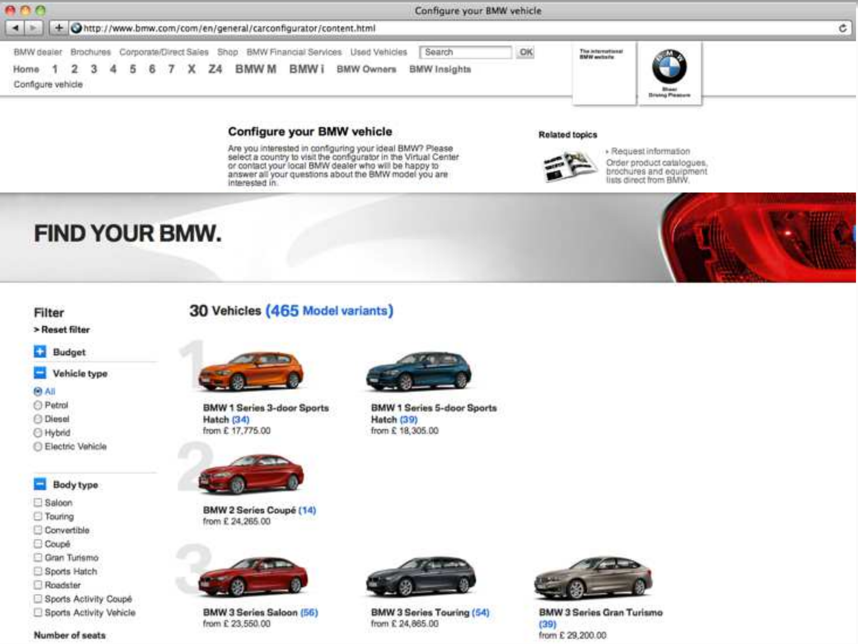
- end-user visible pieces of functionality that represent both commonalities (e.g. mandatory, required) and variabilities (e.g. optional, alternative)
- only specific feature combinations concern valid products

"We always have 126,000,000 different bicycles in store! But only the parts for 1,000..."

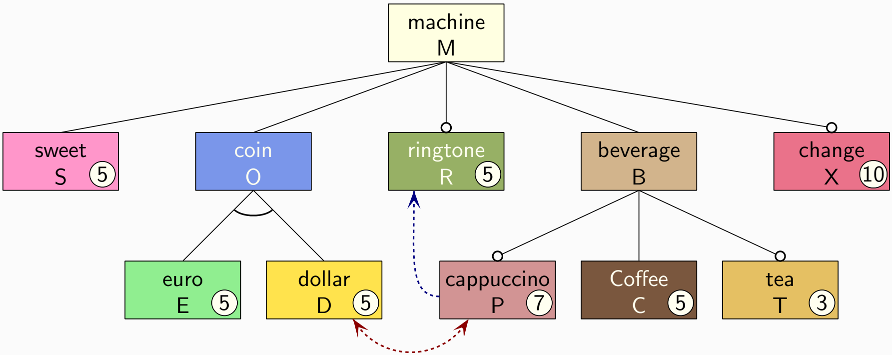
SPL Example (1/2)



SPL Example (2/2)



Attributed feature model: representation of all products

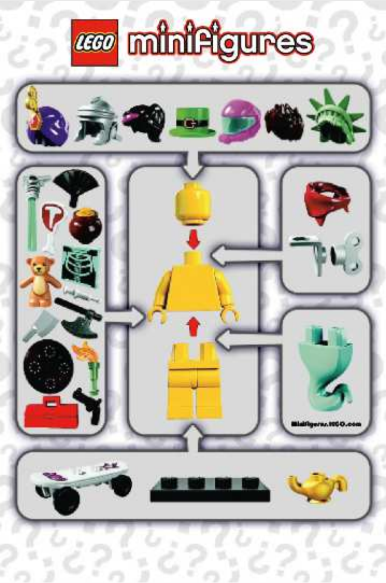


total cost ≤ 30 $\qquad$ $\mathit{cost}\,(\mathit{product}) = \sum\{\, \mathit{cost}\,(\mathit{feature}) \mid \mathit{feature} \in \mathit{product}\,\}$

(from $2^{10} - 1 \xrightarrow{\text{feature diagram}} 2^5 \xrightarrow{\text{cross-tree constraints}} 20 \xrightarrow{\text{attributes}} 16$ valid products)

family of 16 valid products
(i.e. feature combinations)

'feature model'
(allowing >16)



Formal modelling and verification in SPLE

computer-aided analysis of variability models

- traditionally focus on modeling/analyzing structural constraints
- but software systems often embedded/distributed/safety-critical
- important to model/verify also behavior (e.g. quality assurance)

goal: rigorously establish critical requirements of (software) systems

lift success stories from single product/system engineering to SPLE

recent approaches to formally model behavioral variability:

- variants of UML diagrams (Haugen et al., Jézéquel et al.)
- extensions of Petri nets (Clarke et al.)
- variety of models with LTS-like semantics: MTS (Fischbein et al., Fantechi et al.), I/O automata (Larsen et al., Lauenroth et al.), CCS/CSP/CCP/SCCP (Gruler et al., Gnesi et al., ter Beek et al., Tribastone), FTS (Classen et al.), FSM (Millo et al.), etc...

Scalability is a major issue! (slide by C. Kästner, CMU)

with 33 optional, independent features

a unique product for every

person on this planet

Tooling for behavioral analysis of SPLs

Recommendations for the Usability of
Formal Methods for Product Lines (Atlee et alii, FormaliSE'13)

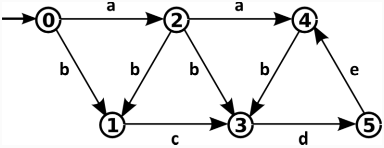
- *adopt and extend state-of-the-art analysis tools*
- *examine[s] only valid product variants*
- *visualize and (manually or automatically) analyze feature combinations corresponding to products of the product line*
- *support (feature) modularity*

Fisler & Krishnamurthi (ESEC/FSE'01) first recognized that most properties of interest naturally decompose around features

mCRL2: language and toolset

- formal, process-algebraic specification of distributed and concurrent systems, associated industrial-strength toolset
- exploration of 10^6 states/s, state spaces up to 10^{12} states
- built-in datatypes, user-defined abstract datatypes, parametrized actions
- modal μ-calculus with data
- visualization, behavioral reduction, model checking
- highly optimized, actively maintained
- intermediate artifacts user-accessable

`www.mcrl2.org`

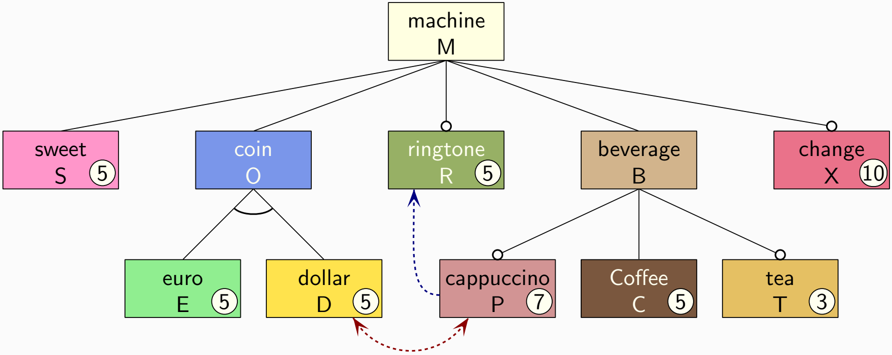



```
proc Foo(st:Int) =
  ( st==0 ) -> ( b.Foo(1) + a.Foo(2) ) +
  ( st==1 ) -> ( c.Foo(3) ) +
  ( st==2 ) -> ( b.Foo(1) + b.Foo(3) + a.Foo(4) ) +
      ...
```

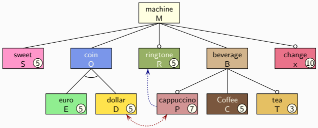
- `[true*]<true>true`: absence of deadlock
- `[true*.b.true*.a] false`: no a after a b
- `mu Y.(<true>true && [!c] Y)`:
  eventually c can be done (or deadlock occurs earlier)
- `mu Y.((nu Z.(<b.d.e> Z)) || [true] Y)`:
  eventually, an infinite sequence of bde is possible

A product line or family of coffee machines

- initially, money must be inserted: either at least one euro's worth in coins, *exclusively* for European products, or at least one dollar's worth in coins, *exclusively* for Canadian products
- input of money can be canceled via a cancel button; *optionally*, the machine returns change if more than one euro or one dollar was inserted
- once the machine contains at least one euro or one dollar, the user chooses whether (s)he wants sugar, by pressing one of two buttons, after which (s)he can select a beverage
- the choice of beverage (coffee, tea, cappuccino) varies, but coffee *must* be offered by all products, whereas cappuccino *may* be offered *solely* by European products
- *optionally*, a ringtone *may* be rung after delivering a beverage; a ringtone *must* be rung by *all* products offering cappuccino
- after the beverage is taken, the machine returns idle



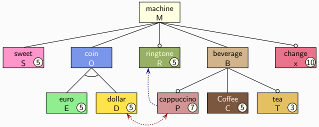
total cost ≤ 30



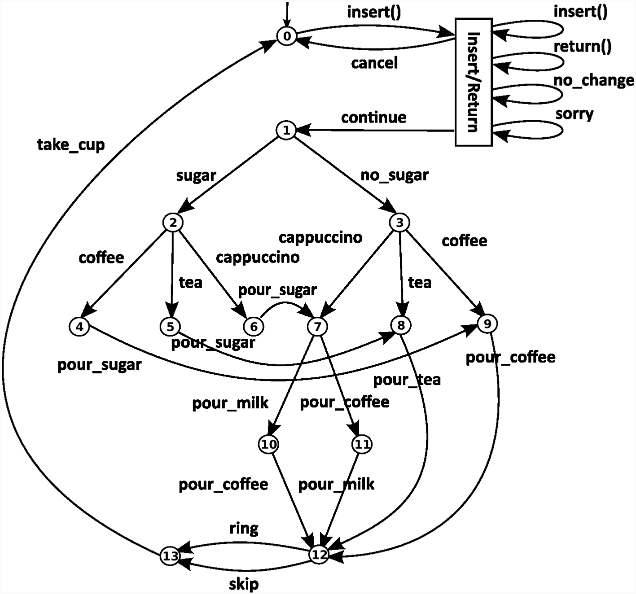
initial call `Sel(0,[M])`

```
proc Sel(st:Int,fs:FSet) =
   ...
( st == 1 ) -> (
   ( M in fs ) -> (
     setO . Sel(2,insert(O,fs) )
) ) +
( st == 2 ) -> (
   ( M in fs ) -> (
     tau . Sel(3,fs) +
     setR . Sel(3,insert(R,fs) )
) ) + ...
```

mCRL2 process Sel (continued)

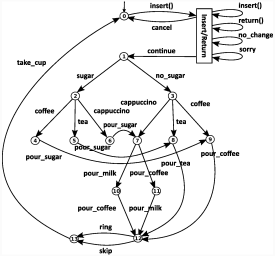



```
    ...
  ( st == 8 ) -> (
  ( ( D in fs ) && ( P in fs ) ) ->
    wrong_set . delta <>
  ( !( R in fs ) && ( P in fs ) ) ->
    wrong_set . delta <>
  ctc_ok . Sel(9,fs)
) +
( st == 9 ) -> (
  ( tcost(fs) <= 30 ) ->
    set_ok(fs) . cost( tcost(fs) ) . Prod(0,fs) <>
  wrong_set . delta );
```



FTS feature labels omitted for readability

mCRL2 process Prod




```
proc Prod(st:Int,fs:FSet) =
  ( st == 0 ) -> ( Insert(0,fs) ) + ...
    ...
  ( st == 2 ) -> (
    ( C in fs ) -> coffee . Prod(4,fs) +
    ( T in fs ) -> tea . Prod(5,fs) +
    ( P in fs ) -> cappuccino . Prod(6,fs)
  ) + ...
```

mCRL2 process Insert

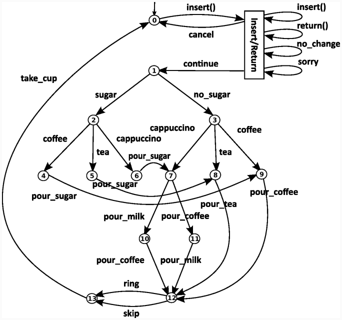
```
proc Insert(bal:Nat,fs:FSet) =
( bal < 100 ) -> (
  ( D in fs ) -> (
    insert(dime) . Insert(bal+10,fs) +
    insert(quarter) . Insert(bal+25,fs) +
    insert(half) . Insert(bal+50,fs) +
    insert(dollar) . Insert(bal+100,fs) ) +
  ( E in fs ) -> (
    insert(ct10) . Insert(bal+10,fs) +
    insert(ct20) . Insert(bal+20,fs) +
    insert(ct50) . Insert(bal+50,fs) +
    insert(euro) . Insert(bal+100,fs) ) ) +
( ( bal > 0) && ( bal < 100 ) ) ->
  Return(bal,fs) . cancel . Prod(0,fs) +
( bal >= 100 ) -> (
  ( ( !(X in fs) ) ->
    no_change . continue . Prod(1,fs) <>
  Return(Int2Nat(bal-100),fs) .
    continue . Prod(1,fs) ) );
```

Example properties

- ```
  [(!continue)*.take_cup] false
  ```
  if payment is not settled, no beverage is delivered
- ```
  [true*.setX.true*.no_change] false
  ```
  once feature X is selected, no_change will not occur
- ```
  [true*] forall fs:FSet.
      <set_ok(fs)> (val((D in fs) => !(P in fs)))
  ```
  if a product is configured successfully, then
  it cannot accept dollars and also provide cappuccino
- ```
  mu Y.(<exists fs:FSet.set_ok(fs)> true
      || <wrong_set> true || [true] Y)
  ```
  eventually either set_ok or wrong_set can occur
- ```
  forall c:Coin.[true*.insert(c)]
      mu Y.(<cancel||take_cup> true || [true] Y)
  ```
  after money is inserted, eventually a beverage is given
  or money insertion is canceled
  ```

A component-based approach

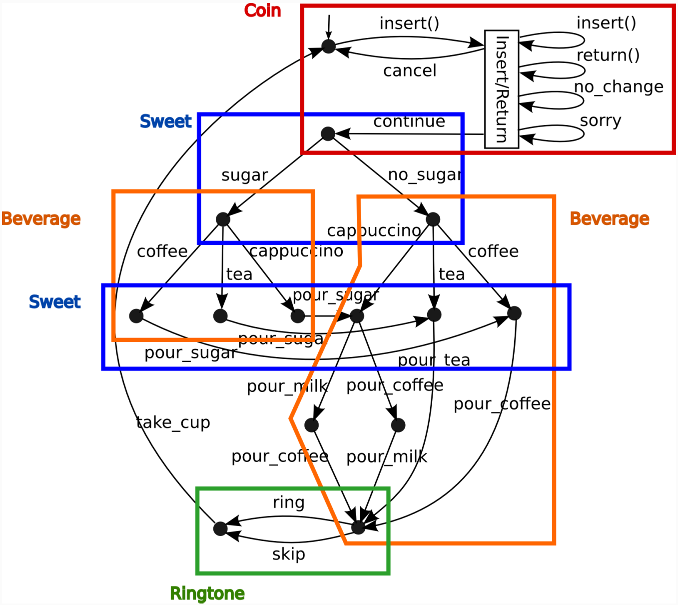
Decomposition of product behaviour



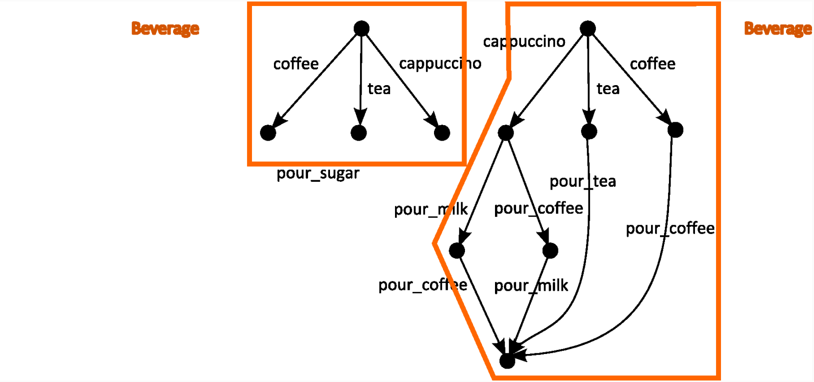
five main components

- Machine (m): take_cup
- Sweet (s): sugar, no_sugar, pour_sugar
- Coin/Change (o,d,e,x): insert, cancel, no_change, sorry, continue
- Ringtone (r): ring, skip
- Beverage (b,c,t,p): coffee, tea, cappuccino, pour_coffee, pour_tea, pour_milk

Decomposition of product behaviour (continued)



Isolated Beverage component



Driver and component processes

```
proc Driver(st:Int) =
  drv_start(st) . sum st':Int . catch(st') . Driver(st');

proc Beverage(fs:FSet) =
  cmp_start(2) . (
    ( C in fs ) -> coffee . raise(4) . Beverage() +
    ( T in fs ) -> tea . raise(5) . Beverage() +
    ( P in fs ) -> cappuccino . raise(6) . Beverage() ) +
    ...
  cmp_start(10) .
    pour_coffee . raise(12) . Beverage() +
  cmp_start(11) .
    pour_milk . raise(12) . Beverage();
```

Driver, component(s) and stub process

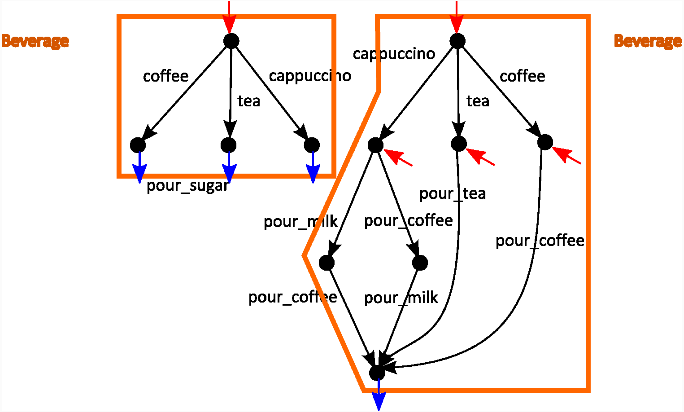
transforming

```
    Driver(0) || Sweet(fs) || Coin(fs) ||
        Ringtone(fs) || Beverage(fs) || Machine
```

into

```
    Driver(0) || Beverage(fs) || Stub
```

```
proc Stub =
  cmp_start(0)  . other . ( raise(2) + raise(3) ) . Stub +
  cmp_start(4)  . other . raise(9) . Stub +
  cmp_start(5)  . other . raise(8) . Stub +
  cmp_start(6)  . other . raise(7) . Stub +
  cmp_start(12) . other . ( raise(2) + raise(3) ) . Stub;
```



(re)entry and exit points

Model checking with stubs

- example property (modal μ-formula in CTL* – next operator)

  `[ true*.coffee ]( mu X. [ !pour_coffee ] X )`

  after a `coffee` action, a `pour_coffee` action happens within a finite number of steps

- model check property against reduced model

  `Driver(0) || Beverage(fs) || Stub`

- verify branching bisimilarity of reduced model

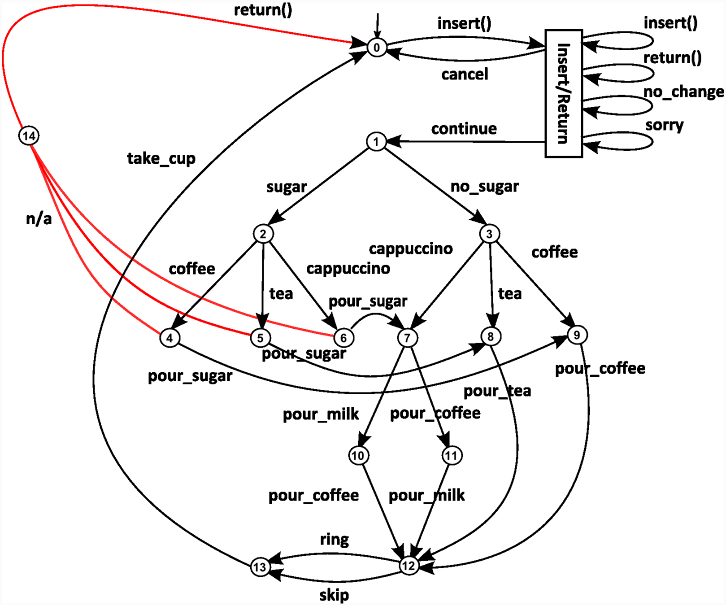
  `Driver(0) || Beverage(fs) || Stub`

  and full model (with `ltscompare` tool in the mCRL2 toolset)

  ```
  Driver(0) || Sweet(fs) || Coin(fs) ||
      Ringtone(fs) || Beverage(fs) || Machine )
  ```

De Nicola & Vaandrager, J. ACM 42 (1995) 458–487

A variation


return()
insert()
insert()
return()
no_change
sorry
Insert/Return
cancel
continue
take_cup
sugar
no_sugar
n/a
cappuccino
coffee
cappuccino
coffee
tea
tea
pour_sugar
pour_sugar
pour_sugar
pour_coffee
pour_tea
pour_milk
pour_coffee
pour_coffee
pour_milk
ring
skip
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Relevant components

previous example property

```
[ true*.coffee ]( mu X. [ !pour_coffee ] X )
```

new example property

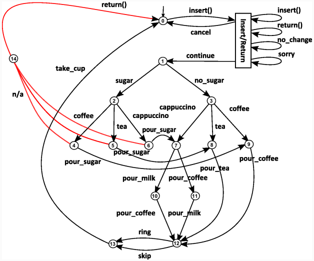
```
[true*.coffee]( mu X.[ !(pour_coffee||return) ]X )
```

action `return` does not belong to component `Beverage`

will check adapted property

```
[true*.coffee]( mu X.[ !(pour_coffee||escape) ]X )
```

action `escape` represents pre-emption of component




```
proc Stub2 =
  cmp_start(0) . other . (raise(2) + raise(3) ) . Stub2 +
  cmp_start(4) . (
    escape . ( raise(2) + raise(3) ) . Stub2 +
    other . raise(9) . Stub2 ) +
  cmp_start(5) . ( ... ) +
  cmp_start(6) . ( ... ) +
  cmp_start(12) . other . ( raise(2) + raise(3) ) . Stub2 ;
```

Model checking with stubs (revisited)

verification of adapted property `adapted_prop`

`[true*.coffee]( mu X.[ !(pour_coffee || escape) ] X )`

against new reduced model `RedSys`

`Driver(0) || Beverage(fs) || Stub2`

conditional to equivalence of satisfaction

`FullSys` $\models$ `new_prop` vs. `RedSys` $\models$ `adapted_prop`

Extend LTS branching bisimulation to FTS

Featured Transition System (Classen et alii, ICSE'10)

$\mathcal{F}$: finite non-empty set of features, $\mathcal{P} \subseteq \mathbf{2}^{\mathcal{F}}$ subset of products

$\mathbb{B}(\mathcal{F})$: set of boolean expressions over $\mathcal{F}$ (feature expressions)

(S, θ, s_0): featured transition system with set S of states, transition constraint function $\theta \colon S \times \mathcal{A} \cup \{\tau\} \times S \to \mathbb{B}(\mathcal{F})$, initial state $s_0 \in S$

we write $s \xrightarrow{\alpha|\psi} s'$ if $\theta(s, \alpha, s') = \psi$

product $P \in \mathcal{P}$ satisfies feature expression $\varphi \in \mathbb{B}(\mathcal{F})$, denoted $P \models \varphi$, if φ is valid when the Boolean variables corresponding to the features of P are assigned value *true* and those not in P value *false*

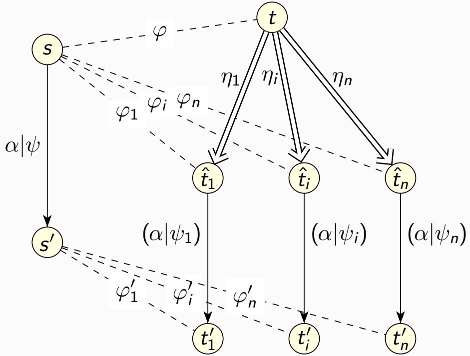
equivalence relation $\sim_{\mathcal{P}}$ on $\mathbb{B}(\mathcal{F})$ given by $\varphi \sim_{\mathcal{P}} \psi$ iff $\forall P \in \mathcal{P}$: $P \models \varphi \Leftrightarrow P \models \psi$; we denote $\widehat{\mathbb{B}}(\mathcal{F}) = \mathbb{B}(\mathcal{F})/{\sim_{\mathcal{P}}}$

Branching feature bisimulation $R \subseteq S \times \widehat{\mathbb{B}}(\mathcal{F}) \times S$

for $s, s' \in S$ and satisfiable $\eta \in \mathbb{B}(\mathcal{F})$ we write $s \stackrel{\eta}{\Rightarrow} s'$ if

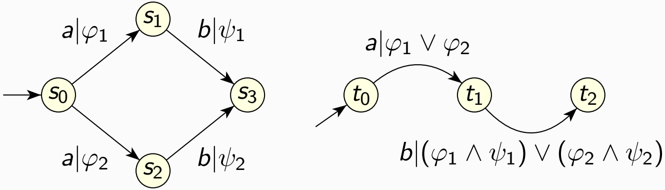
$$\exists n \, \exists s_0, \ldots, s_n \, \exists \eta_1, \ldots, \eta_n :$$
$$s = s_0 \wedge \forall i, 1 \leqslant i \leqslant n \colon s_{i-1} \xrightarrow{\tau|\eta_i} s_i \wedge s' = s_n \wedge \eta = \textstyle\bigwedge_{1 \leqslant i \leqslant n} \eta_i$$

we write $s \xrightarrow{(\alpha|\psi)} s'$ in case $s \xrightarrow{\alpha|\psi} s'$ or $\alpha = \tau \wedge s = s' \wedge \psi = true$

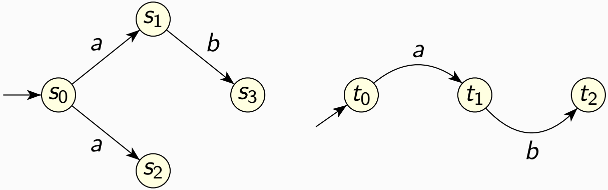


FTS vs. LTS bisimulation

bisimilar FTS assuming $\varphi_1 \wedge \varphi_2 \;\Rightarrow\; (\psi_1 \Leftrightarrow \psi_2)$



non-bisimilar LTS assuming $\varphi_1 \wedge \varphi_2 \wedge \psi_1 \wedge \neg\psi_2$

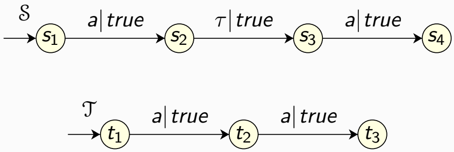


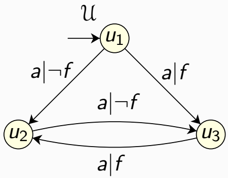
Theorem

Let s, t be states of FTS $\mathcal{S}$. Then $s \simeq_{bf} t$ iff $s \simeq_P t$ for all $P \in \mathcal{P}$.

Minimization modulo branching feature bisimulation

$\mathcal{T}$ and $\mathcal{U}$ are both branching feature bisimilar to $\mathcal{S}$ and both have the minimal number of states, but $\mathcal{U}$ has twice as many transitions as $\mathcal{T}$





Theorem

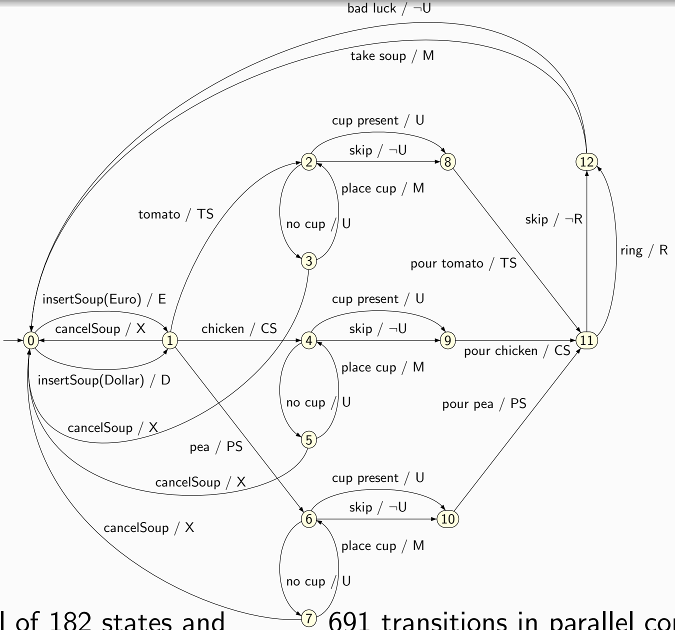
Constructing a minimal branching feature bisimilar FTS is NP-complete.

Theorem

Our polynomial-time sub-optimal algorithm reduces an FTS $\mathcal{S}$ to the smallest FTS $\mathcal{S}_{min}$ for which there exists a coherent branching feature bisimulation relation for $\mathcal{S}$ and $\mathcal{S}_{min}$.

Adaptation of efficient Groote & Vaandrager algorithm (ICALP'90)

Experimental evaluation (beverage + soup component)



(for a total of 182 states and 691 transitions in parallel composition)

mCRL2 results of experimental evaluation (time in seconds)

PROPER-TIES	PRODUCT-BY-PRODUCT				FTS-BASED FAMILY APPROACH			
	WITHOUT BISIMULATION		WITH BISIMULATION		WITHOUT BISIMULATION		WITH BISIMULATION	
	TIME (S)	RESULT	TIME (S)	RESULT	TIME (S)	RESULT	TIME (S)	RESULT
1	42.04	*false*	38.18	*true*	52.96	*false*	13.60	*true*
2	41.78	*true*	41.65	*true*	53.86	*true*	53.69	*true*
3a	42.32	*true*	37.76	*true*	70.57	*true*	7.70	*true*
3b	42.01	*true*	37.78	*true*	59.96	*true*	7.98	*true*
4a	40.62	*true*	38.00	*true*	24.18	*true*	8.65	*true*
4b	40.20	*true*	37.88	*true*	20.78	*true*	10.68	*true*
5a	42.38	*false*	38.51	*true*	66.08	*false*	18.59	*true*
5b	42.34	*false*	38.09	*true*	69.95	*false*	14.92	*true*
6	43.63	*true*	39.17	*true*	105.35	*true*	29.72	*true*
7a	42.45	*true*	38.19	*true*	71.07	*true*	13.84	*true*
7b	42.35	*true*	38.04	*true*	79.05	*true*	9.48	*true*
8	42.82	*true*	39.09	*true*	80.69	*true*	20.47	*true*
TOT	504.94		462.34		754.50		209.32	

1. if a coffee is ordered, then eventually coffee is poured:
   `[ true*.coffee ]( mu X. [ !pour_coffee ] X )`
2. the SPL is deadlock-free: `[true*]<true>true`

etc.

Conclusions and future work

- mCRL2 for software product line analysis
- model reduction before model checking

- automate property and stub adaptation?

- establish subset of modal μ-calculus preserved by branching feature bisimulation
- reduce complexity of minimization algorithm

- does modular verification approach scale to realistic SPL?

`www.mcrl2.org`