Combining Declarative and Procedural Views in the Feature-Oriented Specification and Analysis of Product Families

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy

joint work with

Alberto Lluch Lafuente
IMT, Lucca, Italy

Marinella Petrocchi
IIT–CNR, Pisa, Italy

CINA: 2nd general meeting

Bologna, Italy
18 February 2014

Outline

(Software) Product Line Engineering

Paradigm

To develop a family of products (product line) using a common platform (architecture) and mass customization

Aim

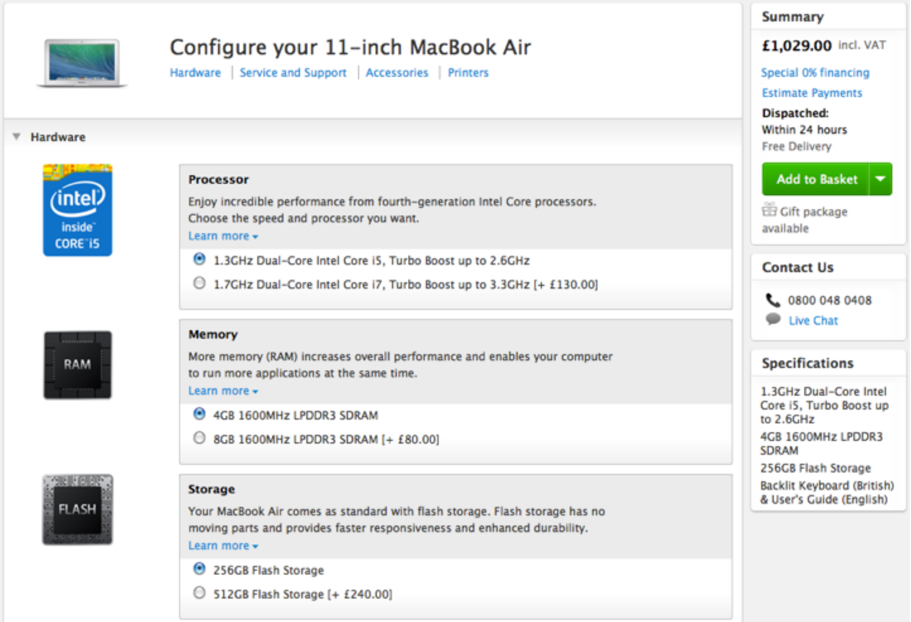
To lower production costs of the individual products by

- letting them share an overall reference model of the product family
- allowing them to differ w.r.t. particular characteristics to serve, e.g., different markets

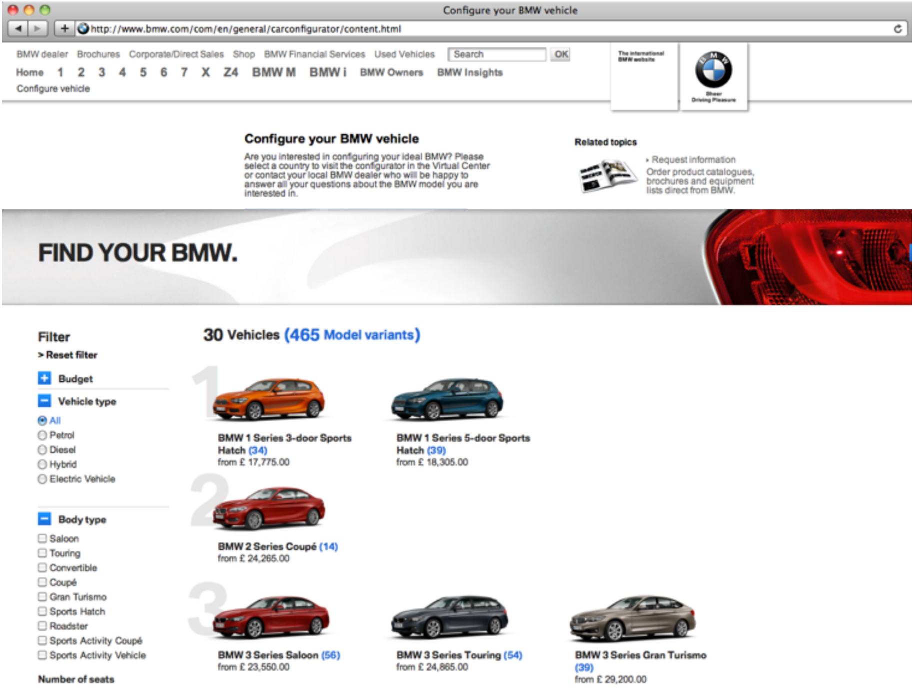
Product variants can be derived from a product family, thus allowing for reuse and differentiation

Production process

Maximize commonalities of product whilst minimizing cost of variations



Configure your 11-inch MacBook Air
Hardware | Service and Support | Accessories | Printers
Hardware
intel inside CORE i5
Processor
Enjoy incredible performance from fourth-generation Intel Core processors.
Choose the speed and processor you want.
Learn more
1.3GHz Dual-Core Intel Core i5, Turbo Boost up to 2.6GHz
1.7GHz Dual-Core Intel Core i7, Turbo Boost up to 3.3GHz [+ £130.00]
RAM
Memory
More memory (RAM) increases overall performance and enables your computer
to run more applications at the same time.
Learn more
4GB 1600MHz LPDDR3 SDRAM
8GB 1600MHz LPDDR3 SDRAM [+ £80.00]
FLASH
Storage
Your MacBook Air comes as standard with flash storage. Flash storage has no
moving parts and provides faster responsiveness and enhanced durability.
Learn more
256GB Flash Storage
512GB Flash Storage [+ £240.00]
Summary
£1,029.00 incl. VAT
Special 0% financing
Estimate Payments
Dispatched:
Within 24 hours
Free Delivery
Add to Basket
Gift package
available
Contact Us
0800 048 0408
Live Chat
Specifications
1.3GHz Dual-Core Intel
Core i5, Turbo Boost up
to 2.6GHz
4GB 1600MHz LPDDR3
SDRAM
256GB Flash Storage
Backlit Keyboard (British)
& User's Guide (English)


Configure your BMW vehicle
http://www.bmw.com/com/en/general/carconfigurator/content.html
BMW dealer Brochures Corporate/Direct Sales Shop BMW Financial Services Used Vehicles
Search OK
Home 1 2 3 4 5 6 7 X Z4 BMW M BMW i BMW Owners BMW Insights
Configure vehicle
Configure your BMW vehicle
Are you interested in configuring your ideal BMW? Please select a country to visit the configurator in the Virtual Center or contact your local BMW dealer who will be happy to answer all your questions about the BMW model you are interested in.
Related topics
Request information
Order product catalogues, brochures and equipment lists direct from BMW.
FIND YOUR BMW.
Filter
> Reset filter
Budget
Vehicle type
All
Petrol
Diesel
Hybrid
Electric Vehicle
Body type
Saloon
Touring
Convertible
Coupé
Gran Turismo
Sports Hatch
Roadster
Sports Activity Coupé
Sports Activity Vehicle
Number of seats
30 Vehicles (465 Model variants)
BMW 1 Series 3-door Sports Hatch (34)
from £ 17,775.00
BMW 1 Series 5-door Sports Hatch (39)
from £ 18,305.00
BMW 2 Series Coupé (14)
from £ 24,265.00
BMW 3 Series Saloon (56)
from £ 23,550.00
BMW 3 Series Touring (54)
from £ 24,865.00
BMW 3 Series Gran Turismo (39)
from £ 29,200.00

Variability analysis

Feature modeling and selection

Provide compact representations of all the products of a product family in terms of their *features* (end-user visible pieces of functionality)

Typically only a subset of feature combinations is valid

Features represent commonalities and variabilities

How to explicitly define **optional**, **alternative**, **mandatory**, **required**, or **excluded** features of a product family as variation points

Managing variability with formal methods

Show that a certain product belongs to a product family or—instead—derive a product from a family by properly selecting features

Formally prove characteristics of products and families alike

(Software) products and product lines

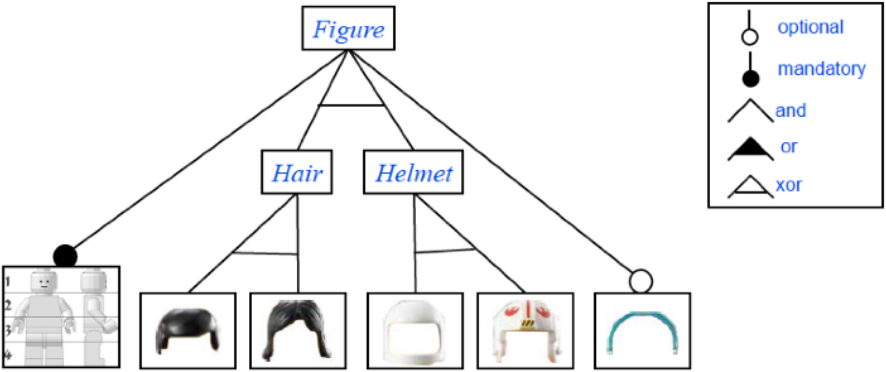
Product

Valid feature combination (configuration)

Product line

Set of valid feature combinations of a domain

Feature model



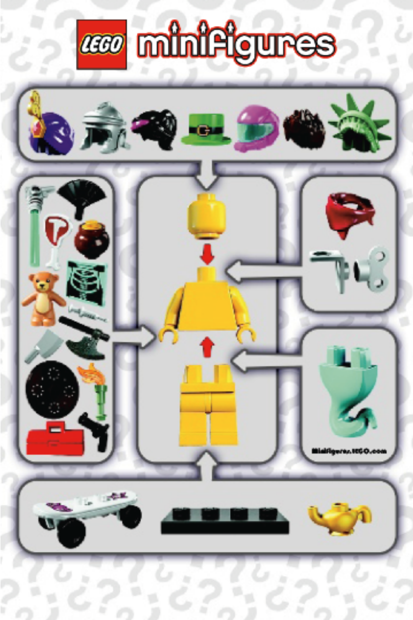
Visor $\Rightarrow$ *Helmet*

Feature model = feature diagram + cross-tree constraints

Many formal (algebraic) approaches ignore cross-tree constraints (e.g. choice calculus, Erwig et al.)

16 valid feature combinations

‘feature model’ (allowing >16)



Formal methods in SPLE

Aim

- Traditionally: focus on modelling/analysing structural constraints
- But: software systems often embedded/distributed/safety critical
- Important: model/analyse also behaviour (e.g. quality assurance)

Or, in the words of Dave Clarke (Uppsala University, Sweden)

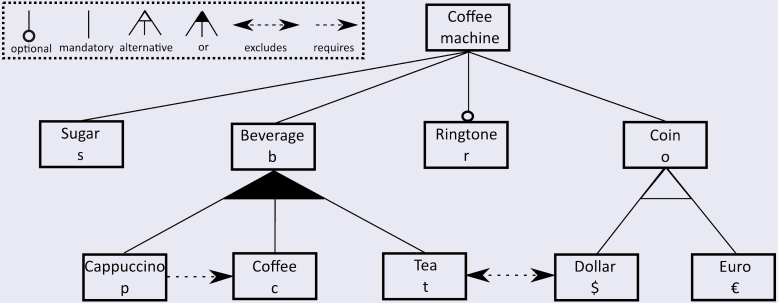
"Behaviour is what we need. Without behaviour, it's just sticks and balls. With behaviour, you get cricket."

Since 2006 several approaches

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

Running example: Family of coffee machines

Structural constraints: Feature Model



Behavioural constraints

- **Initially** a coin must be inserted, **after** which the user must choose whether s/he wants sugar, **after** which s/he may select a beverage
- A ringtone must be rung **after** serving cappuccino, otherwise it may
- The coffee machine returns idle **after** the beverage has been taken

FLAN: Feature-oriented Language

Considers both structural and behavioural constraints

- *Concurrent constraint programming paradigm* as applied in calculi
- Implemented in Maude (Buscemi et al.)

Combines declarative and procedural specification

- A store of constraints allows specifying all common structural constraints from feature models (incl. cross-tree) in a *declarative* way
- A rich set of process-algebraic operators allows specifying both the configuration and behaviour of products in a *procedural* way
- Semantics neatly unifies static and dynamic feature selection

Declarative and procedural views closely related

1. process execution is constrained by store to avoid inconsistencies
2. process can query store to resolve configuration/behavioural option
3. process can update store by adding new features

FLAN: Syntax

With actions $a \in \mathcal{A}$, propositions $p \in \mathcal{P}$ and features $f, g \in \mathcal{F}$

$$
\begin{array}{lrcl}
\text{(fragments)} & F & ::= & [S \mid P] \\
\text{(constraints)} & S, T & ::= & K \mid f \triangleright g \mid f \otimes g \mid S\ T \mid \top \mid \bot \\
\text{(processes)} & P, Q & ::= & \mathit{0} \mid X \mid A.P \mid P + Q \mid P; Q \mid P \parallel Q \\
\text{(actions)} & A & ::= & \mathrm{install}(f) \mid \mathrm{ask}(K) \mid a \\
\text{(propositions)} & K & ::= & p \mid \neg K \mid K \vee K
\end{array}
$$

Constraints

- Store: *consistent(S)*, inconsistent ($\bot$) or no constraint at all ($\top$)
- Universe $\mathcal{P}$ of propositions: predicates *has*(f) and *in(context)*
- Action constraints $do(a) \rightarrow p$: guard to allow/forbid executing a

Processes

$\mathit{0}$:	empty process that can do nothing	X :	process identifier
$A.P$:	process willing to perform action A and then to behave as P		
$P+Q$:	process that can non-deterministically choose to behave as P or as Q		
$P; Q$:	process that must progress first as P and then as Q		
$P \parallel Q$:	process formed by parallel composition of P and Q, evolving independently		

FLAN: Semantics in SOS style

$\rightarrow \subseteq \mathbb{F} \times \mathbb{F}$, with $\mathbb{F}$ set of all terms generated by F

$$(\textsc{Inst})\ \frac{\mathit{consistent}(S\ \mathit{has}(f))}{[S \mid \mathrm{install}(f).P] \longrightarrow [S\ \mathit{has}(f) \mid P]} \qquad (\textsc{Or})\ \frac{[S \mid P] \longrightarrow [S' \mid P']}{[S \mid P + Q] \longrightarrow [S' \mid P']}$$

$$(\textsc{Ask})\ \frac{S \vdash K}{[S \mid \mathrm{ask}(K).P] \longrightarrow [S \mid P]} \qquad (\textsc{Seq})\ \frac{[S \mid P] \longrightarrow [S' \mid P']}{[S \mid P; Q] \longrightarrow [S' \mid P'; Q]}$$

$$(\textsc{Act})\ \frac{S \vdash (do(a) \rightarrow K) \qquad S \vdash K}{[S \mid a.P] \longrightarrow [S \mid P]} \qquad (\textsc{Par})\ \frac{[S \mid P] \longrightarrow [S' \mid P']}{[S \mid P \parallel Q] \longrightarrow [S' \mid P' \parallel Q]}$$

Modulo structural congruence relation $\equiv \subseteq \mathbb{F} \times \mathbb{F}$

$$\begin{array}{rclrclrcl}
P + (Q + R) & \equiv & (P + Q) + R & P + 0 & \equiv & P & P + Q & \equiv & Q + P \\
P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R & 0; P & \equiv & P & P \parallel Q & \equiv & Q \parallel P \\
P; (Q; R) & \equiv & (P; Q); R & P; 0 & \equiv & P & & & \\
 & & & P \parallel 0 & \equiv & P & P & \equiv & P[^{Q}/_{X}] \text{ if } X \doteq Q
\end{array}$$

Axioms naturally and efficiently treated by Maude

1. semantics is (efficiently) executable
2. correspond 1-1 to conditional rewrite rules in Maude implementation
3. few rules: semantics and implementation compact and easy to read

Running example: A specification

$$
\begin{array}{rcl}
F & \doteq & [S \mid D; R] \\
S & \doteq & S_1\ S_2 \\
S_1 & \doteq & has(euro) \vee has(dollar) \\
 & & in(Europe) \rightarrow has(euro) \quad in(Canada) \rightarrow has(dollar) \\
 & & has(coffee) \vee has(cappuccino) \vee has(tea) \quad has(tea) \rightarrow in(Europe) \\
 & & dollar \otimes euro \quad cappuccino \triangleright coffee \\
 & & do(euro) \rightarrow has(euro) \quad do(dollar) \rightarrow has(dollar) \quad do(tea) \rightarrow has(tea) \\
 & & do(coffee) \rightarrow has(coffee) \quad do(cappuccino) \rightarrow has(cappuccino) \\
 & & do(sugar) \rightarrow has(sugar) \quad do(ringtone) \rightarrow has(ringtone) \\
S_2 & \doteq & in(Europe) \\
 & & has(euro) \quad has(dollar) \\
D & \doteq & \mathrm{install}(sugar).0 \mid \mathrm{install}(coffee).0 \mid \mathrm{install}(tea).0 \mid \mathrm{install}(cappuccino).0 \\
R & \doteq & (\mathrm{ask}(in(Europe)).euro.0 + \mathrm{ask}(in(Canada)).dollar.0); (P_2 + P_3) \\
P_2 & \doteq & sugar.P_3 \\
P_3 & \doteq & coffee.P_4 + tea.P_4 + cappuccino.P_5 \\
P_4 & \doteq & P_5 + R \\
P_5 & \doteq & \mathrm{install}(ringtone).ringtone.R
\end{array}
$$

Declarative and procedural feature configuration

Select feature f in an *explicit* and *declarative* way

- Include the proposition $has(f)$ in the initial store
- For features that are surely mandatory for all the family's products

Select feature f in an *implicit* and *declarative* way

- Derive f as a consequence of other constraints
- For features that apparently seem not to be mandatory, but that are indeed enforced by the constraints (e.g. in a store with both constraints $g \triangleright f$ and $has(g)$, the presence of f can be inferred)

Install feature f dynamically in a *procedural* way

- Install f during the execution of a process
- Allows designer to delay feature configuration decisions to runtime
- This is a key aspect of our concurrent constraint approach

Checking the (in)consistency of the initial constraints

Returns ∅ if consistent, else subset of inconsistent constraints

```
reduce in ANALYSIS-KRIPKE : inconsistency(S) .
...
result neConstraints: has(dollar) has(euro)
                      dollar * euro
```

Specification needs to be corrected

Delegate installation of *euro* and *dollar* to configuration process *D* by invoking **install**(*euro*).*0* and **install**(*dollar*).*0*

Returns true if consistent, else false

```
reduce in ANALYSIS-KRIPKE : consistent(S) .
...
result Bool:  true
```

Running example: Revised specification

$$
\begin{array}{rcl}
F & \doteq & [S \mid D; R] \\
S & \doteq & S_1 \; S_2 \\
S_1 & \doteq & has(euro) \vee has(dollar) \\
 & & in(Europe) \rightarrow has(euro) \quad in(Canada) \rightarrow has(dollar) \\
 & & has(coffee) \vee has(cappuccino) \vee has(tea) \quad has(tea) \rightarrow in(Europe) \\
 & & dollar \otimes euro \quad cappuccino \triangleright coffee \\
 & & do(euro) \rightarrow has(euro) \quad do(dollar) \rightarrow has(dollar) \quad do(tea) \rightarrow has(tea) \\
 & & do(coffee) \rightarrow has(coffee) \quad do(cappuccino) \rightarrow has(cappuccino) \\
 & & do(sugar) \rightarrow has(sugar) \quad do(ringtone) \rightarrow has(ringtone) \\
S_2 & \doteq & in(Europe) \\
 & & \cancel{has(euro)} \quad \cancel{has(dollar)} \\
D & \doteq & \text{install}(sugar).0 \mid \text{install}(coffee).0 \mid \text{install}(tea).0 \mid \text{install}(cappuccino).0 \\
 & & \mid \text{install}(euro).0 \mid \text{install}(dollar).0 \\
R & \doteq & (\text{ask}(in(Europe)).euro.0 + \text{ask}(in(Canada)).dollar.0); (P_2 + P_3) \\
P_2 & \doteq & sugar.P_3 \\
P_3 & \doteq & coffee.P_4 + tea.P_4 + cappuccino.P_5 \\
P_4 & \doteq & P_5 + R \\
P_5 & \doteq & \text{install}(ringtone).ringtone.R
\end{array}
$$

Executing the configuration process

Applies rewrite rules until a fix point is reached

```
rewrite in ANALYSIS-KRIPKE : ! [S | D] .
...
result KFragment: ! [has(dollar)  has(coffee)
has(tea) has(cappuccino) has(sugar) ... | 0]
```

Checking the consistency of all configurations

FLAN's semantics preserves consistency

Still we can use Maude's model checker to check consistency of all reachable configurations by specifying the property $\Box$ *isConsistent* (i.e. consistency is an invariant)

State predicate returns the result of **consistent(S)**

```
reduce in ANALYSIS-KRIPKE : modelCheck( ( !  [ S | D ] ) ,
[] isConsistent ) .
...
result Bool:  true
```

Checking behavioural properties

Check that runtime behaviour does not introduce inconsistencies

```
reduce in ANALYSIS-KRIPKE : modelCheck( ( ! [ S | D ; R ] ) ,
[] isConsistent ) .
...
result Bool: true
```

Check *"a ringtone must be rung after serving a cappuccino"* ...

```
reduce in ANALYSIS-LTS : modelCheck( ( !  (do('machine)
[S' | D' ; R]) ) , [] (cappuccino -> <> ringtone) ) .
...
result Bool:  true
```

... is preserved if we replace procedural by declarative information

The conditional statement used to accept dollar or euro is redundant: A simpler run-time process replaces it with a non-deterministic choice that will be consistently solved at runtime since the store contains the action constraints $do(euro) \rightarrow has(euro)$ and $do(dollar) \rightarrow has(dollar)$ which will forbid the use of actions *euro* or *dollar* if the corresponding feature has not been installed

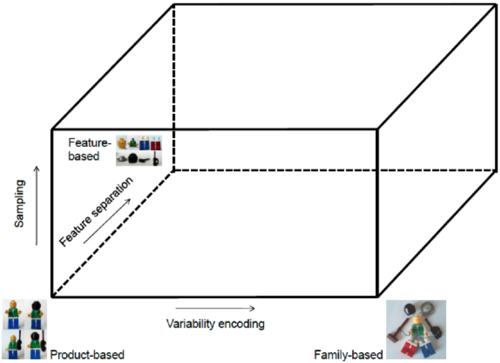
Final specification

$$
\begin{aligned}
F &\doteq [S \mid D; R] \\
S &\doteq S_1\ S_2 \\
S_1 &\doteq has(euro) \vee has(dollar) \\
&\quad in(Europe) \rightarrow has(euro) \quad in(Canada) \rightarrow has(dollar) \\
&\quad has(coffee) \vee has(cappuccino) \vee has(tea) \quad has(tea) \rightarrow in(Europe) \\
&\quad dollar \otimes euro \quad cappuccino \triangleright coffee \\
&\quad do(euro) \rightarrow has(euro) \quad do(dollar) \rightarrow has(dollar) \quad do(tea) \rightarrow has(tea) \\
&\quad do(coffee) \rightarrow has(coffee) \quad do(cappuccino) \rightarrow has(cappuccino) \\
&\quad do(sugar) \rightarrow has(sugar) \quad do(ringtone) \rightarrow has(ringtone) \\
S_2 &\doteq in(Europe) \\
D &\doteq \mathrm{install}(sugar).0 \mid \mathrm{install}(coffee).0 \mid \mathrm{install}(tea).0 \mid \mathrm{install}(cappuccino).0 \\
&\quad \mid \mathrm{install}(euro).0 \mid \mathrm{install}(dollar).0 \\
R &\doteq (\cancel{\mathrm{ask}(in(Europe))}.euro.0 + \cancel{\mathrm{ask}(in(Canada))}.dollar.0); (P_2 + P_3) \\
P_2 &\doteq sugar.P_3 \\
P_3 &\doteq coffee.P_4 + tea.P_4 + cappuccino.P_5 \\
P_4 &\doteq P_5 + R \\
P_5 &\doteq \mathrm{install}(ringtone).ringtone.R
\end{aligned}
$$

FLAN's position in the PLA cube (Apel et al.)

Sampling based on coverage criteria such as pairwise or t-wise coverage (or other heuristics)

Feature interactions!



Family-based analysis: check properties of entire product family

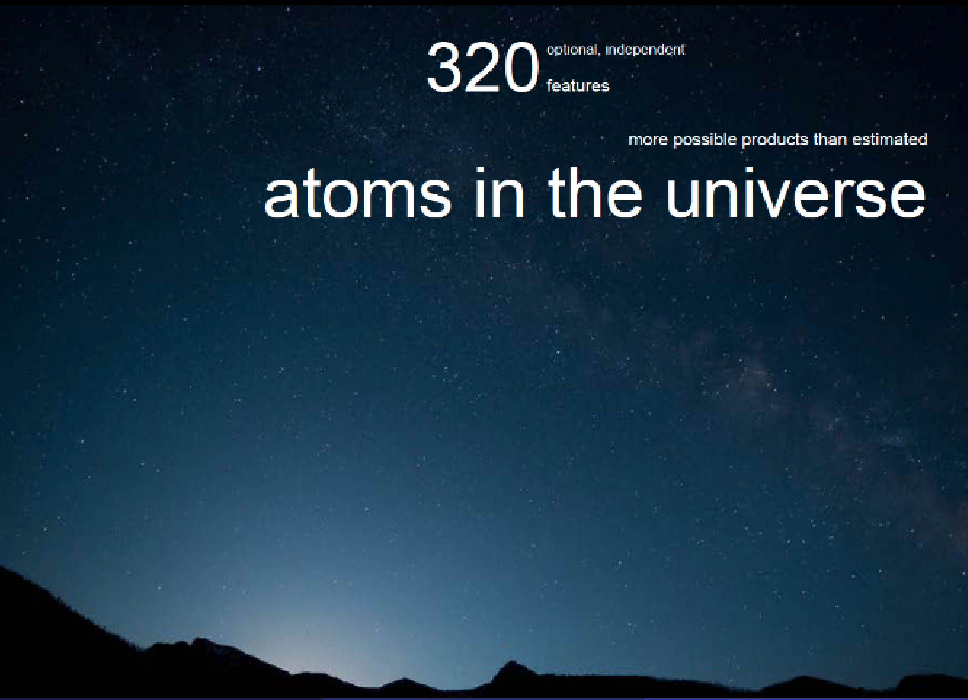
In general checks like $[S \mid P] \models \phi$: does $[S \mid P]$ satisfy LTL property ϕ?
A positive result means the whole family specified by $[S \mid P]$ satisfies ϕ
A negative result—instead—witnesses that *at least one* of its products has at least one behaviour that does not satisfy ϕ

Ongoing work on further interesting analyses

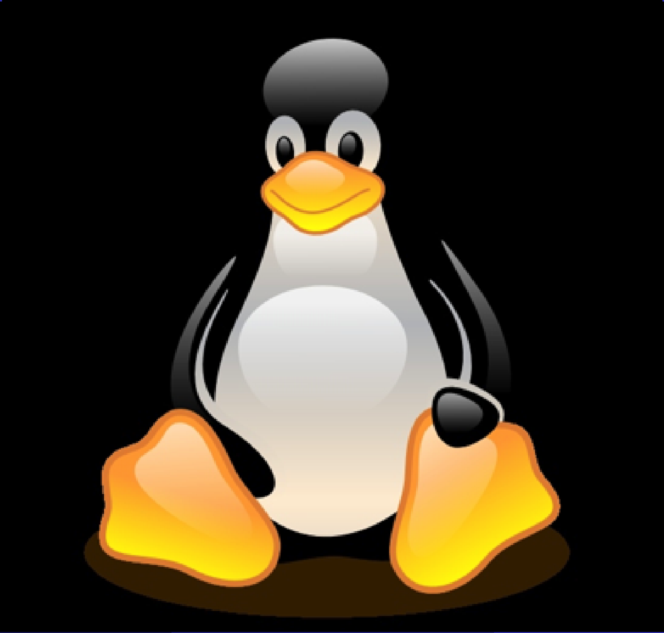
Aim: identify subclasses of configurations satisfying specific properties with 33 optional, independent features

a unique product for every

person on this planet


320 optional, independent features
more possible products than estimated
atoms in the universe

Linux: 10,000 configurable options. . .



Conclusions

Feature-oriented Language FLAN

- Proof of concept for specifying and analysing both declarative and procedural aspects of product families
- Its semantics neatly unifies static and dynamic feature selection

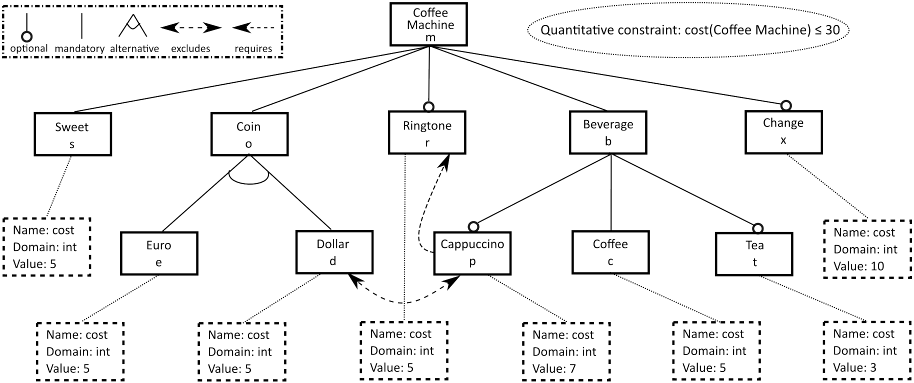
Not *the* language, but useful features to adopt in other languages

- Concurrent constraint programming: flexible mechanism for both declarative *and* procedural aspects (e.g. delay design decisions until runtime, free runtime specifications from feature constraints, thus resulting in light-weight and understandable specifications)

Implementation in Maude: exploit Maude's rich analysis toolset

- For now SAT solver, reachability analyser and LTL model checker
- e.g. statistical model checker PVESTA to evaluate the performance of product families in *stochastic* and *quantitative* variants of FLAN

Attributed feature model



Non-functional feature attributes, e.g. cost

$$cost(product) = \sum\{\, cost(feature) \mid feature \in product \,\}$$

Further reduces number of valid products

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

Future work

We envisage several potentially interesting extensions of FLAN:

1. Adopt further primitives and mechanisms from the concurrent constraint programming tradition

   e.g. the concurrent constraint π-calculus of Buscemi & Montanari provides synchronisation mechanisms typical of mobile calculi (i.e. name passing), a **check** operation to prevent inconsistencies, a **retract** operation to remove constraints from the store and a general framework for *soft* constraints (i.e. not only boolean)

2. Provide an FTS and an MTS semantics of FLAN so that:

   1. FLAN becomes a high-level language for those semantic models
   2. We can exploit the specialised analysis tools developed for them