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

1. Software Product Line Engineering
2. Running example: A family of coffee machines
3. Feature-oriented Language FLAN
4. Declarative versus procedural specification
5. Automated analyses with Maude
6. Conclusions and future work
(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 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.

FIND YOUR BMW.

30 Vehicles (465 Model variants)

1. BMW 1 Series 3-door Sports Hatch (34) from £ 17,775.00
2. BMW 1 Series 5-door Sports Hatch (39) from £ 18,305.00
3. BMW 2 Series Coupé (14) from £ 24,265.00
4. BMW 3 Series Saloon (56) from £ 23,550.00
5. BMW 3 Series Touring (54) from £ 24,865.00
6. 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

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

- 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 $a \in A$, propositions $p \in P$ and features $f, g \in F$

| (fragments) $F$ | ::= | $[S \mid P]$ |
| (constraints) $S, T$ | ::= | $K \mid f \triangleright g \mid f \otimes g \mid S \cdot T \mid \top \mid \bot$ |
| (processes) $P, Q$ | ::= | $0 \mid X \mid A.P \mid P + Q \mid P; Q \mid P \parallel Q$ |
| (actions) $A$ | ::= | install(f) | ask(K) | $a$ |
| (propositions) $K$ | ::= | $p \mid \neg K \mid K \lor K$ |

Constraints

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

Processes

| 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**

→ ⊆ F × F, with F set of all terms generated by F

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(INST)</strong></td>
<td>consistent(S has(f))</td>
</tr>
<tr>
<td>[S</td>
<td>install(f).P] → [S has(f)</td>
</tr>
<tr>
<td><strong>(ASK)</strong></td>
<td>S ⊩ K</td>
</tr>
<tr>
<td>[S</td>
<td>ask(K).P] → [S</td>
</tr>
<tr>
<td><strong>(ACT)</strong></td>
<td>S ⊩ (do(a) → K) S ⊩ K</td>
</tr>
<tr>
<td>[S</td>
<td>a.P] → [S</td>
</tr>
<tr>
<td><strong>(OR)</strong></td>
<td>[S</td>
</tr>
<tr>
<td><strong>(SEQ)</strong></td>
<td>[S</td>
</tr>
<tr>
<td><strong>(PAR)</strong></td>
<td>[S</td>
</tr>
</tbody>
</table>

Modulo structural congruence relation ≡ ⊆ F × F

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>P + (Q + R)</td>
<td>(P + Q) + R</td>
</tr>
<tr>
<td>P + 0</td>
<td>P</td>
</tr>
<tr>
<td>P + Q</td>
<td>Q + P</td>
</tr>
<tr>
<td>P ∥ (Q ∥ R)</td>
<td>(P ∥ Q) ∥ R</td>
</tr>
<tr>
<td>0; P</td>
<td>P</td>
</tr>
<tr>
<td>P ∥ Q</td>
<td>Q ∥ P</td>
</tr>
<tr>
<td>P; (Q; R)</td>
<td>(P; Q); R</td>
</tr>
<tr>
<td>P; 0</td>
<td>P</td>
</tr>
<tr>
<td>P ∥ 0</td>
<td>P</td>
</tr>
<tr>
<td>P</td>
<td>P[Q/x] if X ⊨ Q</td>
</tr>
</tbody>
</table>

**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

\[
F \equiv [S \mid D; R]
\]

\[
S \equiv S_1 S_2
\]

\[
S_1 \equiv \text{has(euro)} \lor \text{has(dollar)}
\]

\[
\text{in(Europe)} \rightarrow \text{has(euro)} \quad \text{in(Canada)} \rightarrow \text{has(dollar)}
\]

\[
\text{has(coffee)} \lor \text{has(cappuccino)} \lor \text{has(tea)} \quad \text{has(tea)} \rightarrow \text{in(Europe)}
\]

\[
dollar \otimes \text{euro} \quad \text{cappuccino} \triangleright \text{coffee}
\]

\[
do(euro) \rightarrow \text{has(euro)} \quad do(dollar) \rightarrow \text{has(dollar)} \quad do(tea) \rightarrow \text{has(tea)}
\]

\[
do(coffee) \rightarrow \text{has(coffee)} \quad do(cappuccino) \rightarrow \text{has(cappuccino)}
\]

\[
do(sugar) \rightarrow \text{has(sugar)} \quad do(ringtone) \rightarrow \text{has(ringtone)}
\]

\[
S_2 \equiv \text{in(Europe)}
\]

\[
\text{has(euro)} \quad \text{has(dollar)}
\]

\[
D \equiv \text{install(sugar).0} \mid \text{install(coffee).0} \mid \text{install(tea).0} \mid \text{install(cappuccino).0}
\]

\[
R \equiv (\text{ask(in(Europe)).euro.0} + \text{ask(in(Canada)).dollar.0}); (P_2 + P_3)
\]

\[
P_2 \equiv \text{sugar}.P_3
\]

\[
P_3 \equiv \text{coffee}.P_4 + \text{tea}.P_4 + \text{cappuccino}.P_5
\]

\[
P_4 \equiv P_5 + R
\]

\[
P_5 \equiv \text{install(ringtone).ringtone}.R
\]
Declarative and procedural feature configuration

Select feature \( f \) in an *explicit* and *declarative* way
- Include the proposition \( \text{has}(f) \) 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 \( \text{has}(g) \), 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 \( \emptyset \) if consistent, else subset of inconsistent constraints

\[
\text{reduce in ANALYSIS-KRIPKE : inconsistency}(S) . \\
\ldots
\]
\[
\text{result neConstraints: has(dollar) has(euro) dollar } \ast \text{ euro}
\]

Specification needs to be corrected

Delegate installation of \textit{euro} and \textit{dollar} to configuration process \( D \) by invoking \texttt{install(euro).0} and \texttt{install(dollar).0}

Returns true if consistent, else false

\[
\text{reduce in ANALYSIS-KRIPKE : consistent}(S) . \\
\ldots
\]
\[
\text{result Bool: true}
\]
Running example: Revised specification

\[ F \triangleq [S \mid D; R] \]
\[ S \triangleq S_1 \; S_2 \]
\[ S_1 \triangleq \text{has}(\text{euro}) \lor \text{has}(\text{dollar}) \]
\[ \text{in}(\text{Europe}) \rightarrow \text{has}(\text{euro}) \; \text{in}(\text{Canada}) \rightarrow \text{has}(\text{dollar}) \]
\[ \text{has}(\text{coffee}) \lor \text{has}(\text{cappuccino}) \lor \text{has}(\text{tea}) \; \text{has}(\text{tea}) \rightarrow \text{in}(\text{Europe}) \]
\[ \text{dollar} \otimes \text{euro} \; \text{cappuccino} \triangleright \text{coffee} \]
\[ \text{do}(\text{euro}) \rightarrow \text{has}(\text{euro}) \; \text{do}(\text{dollar}) \rightarrow \text{has}(\text{dollar}) \; \text{do}(\text{tea}) \rightarrow \text{has}(\text{tea}) \]
\[ \text{do}(\text{coffee}) \rightarrow \text{has}(\text{coffee}) \; \text{do}(\text{cappuccino}) \rightarrow \text{has}(\text{cappuccino}) \]
\[ \text{do}(\text{sugar}) \rightarrow \text{has}(\text{sugar}) \; \text{do}(\text{ringtone}) \rightarrow \text{has}(\text{ringtone}) \]
\[ S_2 \triangleq \text{in}(\text{Europe}) \]
\[ \underline{\text{has}(\text{euro})} \; \underline{\text{has}(\text{dollar})} \]
\[ D \triangleq \text{install}(\text{sugar}).0 \mid \text{install}(\text{coffee}).0 \mid \text{install}(\text{tea}).0 \mid \text{install}(\text{cappuccino}).0 \]
\[ \mid \text{install}(\text{euro}).0 \mid \text{install}(\text{dollar}).0 \]
\[ R \triangleq (\text{ask}(\text{in}(\text{Europe})).\text{euro}.0 + \text{ask}(\text{in}(\text{Canada})).\text{dollar}.0); (P_2 + P_3) \]
\[ P_2 \triangleq \text{sugar}.P_3 \]
\[ P_3 \triangleq \text{coffee}.P_4 + \text{tea}.P_4 + \text{cappuccino}.P_5 \]
\[ P_4 \triangleq P_5 + R \]
\[ P_5 \triangleq \text{install}(\text{ringtone}).\text{ringtone}.R \]
Executing the configuration process

Applies rewrite rules until a fix point is reached

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

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) → has(euro) and do(dollar) → has(dollar) which will forbid the use of actions euro or dollar if the corresponding feature has not been installed.
Final specification

\[ F \triangleq [S \mid D; R] \]

\[ S \triangleq S_1 S_2 \]

\[ S_1 \triangleq \text{has(euro)} \lor \text{has(dollar)} \]

\begin{align*}
\text{in(Europe)} & \rightarrow \text{has(euro)} \quad \text{in(Canada)} \rightarrow \text{has(dollar)} \\
\text{has(coffee)} \lor \text{has(cappuccino)} \lor \text{has(tea)} & \quad \text{has(tea)} \rightarrow \text{in(Europe)} \\
\text{dollar} \otimes \text{euro} & \quad \text{cappuccino} \triangleright \text{coffee} \\
\text{do(euro)} & \rightarrow \text{has(euro)} \quad \text{do(dollar)} \rightarrow \text{has(dollar)} \quad \text{do(tea)} \rightarrow \text{has(tea)} \\
\text{do(coffee)} & \rightarrow \text{has(coffee)} \quad \text{do(cappuccino)} \rightarrow \text{has(cappuccino)} \\
\text{do(sugar)} & \rightarrow \text{has(sugar)} \quad \text{do(ringtone)} \rightarrow \text{has(ringtone)}
\end{align*}

\[ S_2 \triangleq \text{in(Europe)} \]

\[ D \triangleq \text{install(sugar).}0 \mid \text{install(coffee).}0 \mid \text{install(tea).}0 \mid \text{install(cappuccino).}0 \\
\mid \text{install(euro).}0 \mid \text{install(dollar).}0 \]

\[ R \triangleq (\text{ask(in(Europe))}.\text{euro.}0 + \text{ask(in(Canada)).dollar.}0); (P_2 + P_3) \]

\[ P_2 \triangleq \text{sugar}.P_3 \]

\[ P_3 \triangleq \text{coffee}.P_4 + \text{tea}.P_4 + \text{cappuccino}.P_5 \]

\[ P_4 \triangleq P_5 + R \]

\[ P_5 \triangleq \text{install(ringtone).ringtone.R} \]
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 $[S | P] \models \phi$: does $[S | P]$ satisfy LTL property $\phi$?

A positive result means the whole family specified by $[S | P]$ satisfies $\phi$.

A negative result—instead—witnesses that at least one of its products has at least one behaviour that does not satisfy $\phi$.

**Ongoing work on further interesting analyses**

Aim: identify subclasses of configurations satisfying specific properties.
Scalability is a major issue!

with 33 features

optional, independent

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

Quantitative constraint: \( \text{cost}(\text{Coffee Machine}) \leq 30 \)

\[
\text{cost}(\text{product}) = \sum\{ \text{cost}(\text{feature}) \mid \text{feature} \in \text{product} \}
\]

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 $\pi$-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