Combining Declarative and Procedural Views in the 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

FMSPLE 2013

Tokyo, Japan
27 August 2013

Outline

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 our PC chair

"Behaviour is what we need. Without behaviour, it's just sticks and balls. With behaviour, you get cricket." Dave Clarke, June 2013

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.)

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 our PC chair

"Behaviour is what we need. Without behaviour, it's just sticks and balls. With behaviour, you get cricket." Dave Clarke, June 2013

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.)

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 our PC chair

"Behaviour is what we need. Without behaviour, it's just sticks and balls. With behaviour, you get cricket." Dave Clarke, June 2013

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.)

FLAN: Feature-oriented Language

Considers both structural and behavioural constraints

- *Concurrent constraint programming paradigm* as applied in calculi
- Implemented in Maude (like CL4SPL presented at FMSPLE 2012)

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: Feature-oriented Language

Considers both structural and behavioural constraints

- *Concurrent constraint programming paradigm* as applied in calculi
- Implemented in Maude (like CL4SPL presented at FMSPLE 2012)

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: Feature-oriented Language

Considers both structural and behavioural constraints

- *Concurrent constraint programming paradigm* as applied in calculi
- Implemented in Maude (like CL4SPL presented at FMSPLE 2012)

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 $f, g \in \mathcal{F}$

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

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

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		
$P \mid Q$:	process formed by parallel composition of P and Q, evolving independently		

FLAN: Syntax

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

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

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

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		
$P \,|\, Q$:	process formed by parallel composition of P and Q, evolving independently		

FLAN: Syntax

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

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

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

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		
$P \mid Q$:	process formed by parallel composition of P and Q, evolving independently		

FLAN: Semantics in SOS style

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

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

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

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

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

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
P + (Q + R) & \equiv & (P + Q) + R & P + 0 & \equiv & P & P + Q & \equiv & Q + P \\
P \,|\, (Q \,|\, R) & \equiv & (P \,|\, Q) \,|\, R & 0; P & \equiv & P & P \,|\, Q & \equiv & Q \,|\, P \\
P; (Q; R) & \equiv & (P; Q); R & P; 0 & \equiv & P & & & \\
 & & & P \,|\, 0 & \equiv & P & P & \equiv & P[{}^{Q}/_{X}] \text{ if } X \doteq Q
\end{array}$$

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

FLAN: Semantics in SOS style

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

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

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

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

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

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
P + (Q + R) & \equiv & (P + Q) + R & P + 0 & \equiv & P & P + Q & \equiv & Q + P \\
P \,|\, (Q \,|\, R) & \equiv & (P \,|\, Q) \,|\, R & 0; P & \equiv & P & P \,|\, Q & \equiv & Q \,|\, P \\
P; (Q; R) & \equiv & (P; Q); R & P; 0 & \equiv & P & & & \\
 & & & P \,|\, 0 & \equiv & P & P & \equiv & P[^{Q}/_{X}] \text{ if } X \doteq Q
\end{array}$$

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

FLAN: Semantics in SOS style

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

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

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

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

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

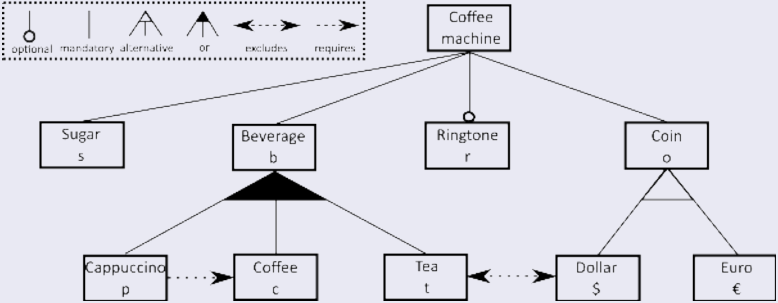
$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
P + (Q + R) & \equiv & (P + Q) + R & P + 0 & \equiv & P & P + Q & \equiv & Q + P \\
P \,|\, (Q \,|\, R) & \equiv & (P \,|\, Q) \,|\, R & 0; P & \equiv & P & P \,|\, Q & \equiv & Q \,|\, P \\
P; (Q; R) & \equiv & (P; Q); R & P; 0 & \equiv & P & & & \\
& & & P \,|\, 0 & \equiv & P & P & \equiv & P[^{Q}/_{X}] \text{ if } X \doteq Q
\end{array}$$

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: family of coffee machines

Structural constraints

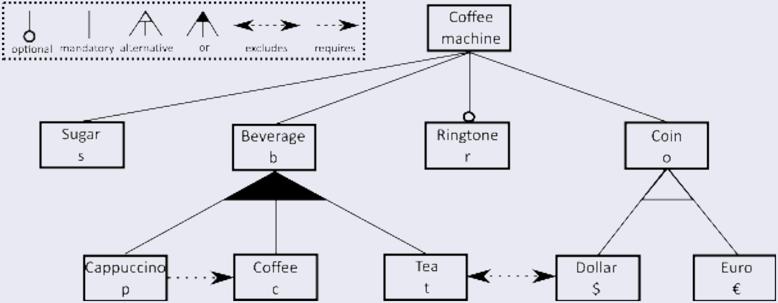


Behavioural constraints

- **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

Running example: family of coffee machines

Structural constraints



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

A specification

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

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

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

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
```

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
```

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
```

Revised specification

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

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 □ *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 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

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

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
```

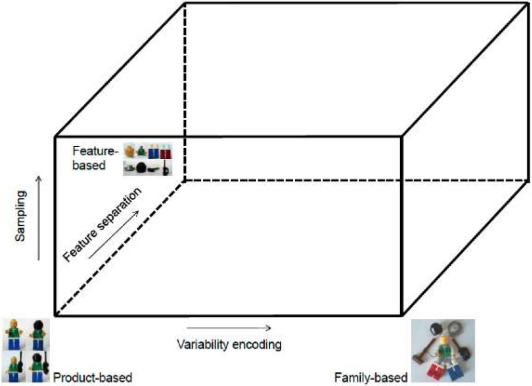
... is preserved if we replace procedural by declarative information

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

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

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



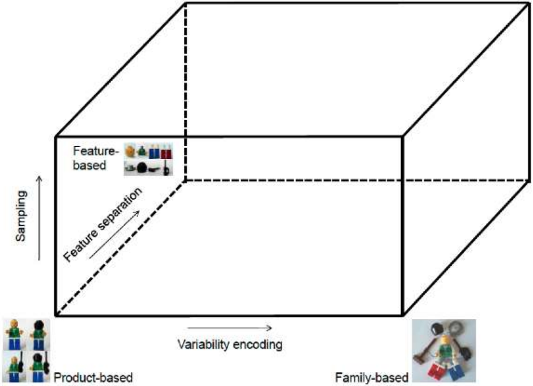
Family-based analysis: check properties of entire product family

In general checks like $[S \parallel P] \models \phi$: does $[S \parallel P]$ satisfy LTL property ϕ?
A positive result means the whole family specified by $[S \parallel P]$ 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

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



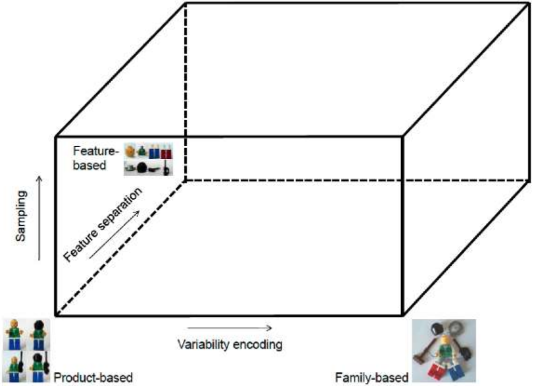
Family-based analysis: check properties of entire product family

In general checks like $[S \parallel P] \models \phi$: does $[S \parallel P]$ satisfy LTL property ϕ?
A positive result means the whole family specified by $[S \parallel P]$ 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

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



Family-based analysis: check properties of entire product family

In general checks like $[S \parallel P] \models \phi$: does $[S \parallel P]$ satisfy LTL property ϕ?
A positive result means the whole family specified by $[S \parallel P]$ 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

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

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

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

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

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

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

Publicity: start working for SPLC 2014 in Florence

http://www.splc2014.net/

Organised by our FMT lab of ISTI–CNR, Pisa