Family-Based Model Checking

Maurice H. ter Beek (ISTI–CNR)

presenting joint work with Erik de Vink and Tim Willemse, TU/e, The Netherlands

3rd QUANTICOL project review
Lucca, May 23rd, 2017
Software product line (SPL) analysis

- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user
Product lines are everywhere

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.

Filter

> Reset filter

- Budget
- Vehicle type
  - All
  - Petrol
  - Diesel
  - Hybrid
  - Electric Vehicle

Body type

- Saloon
- Coupé
- Convertible
- Gran Turismo
- Sports Hatch
- Roadster
- Sports Activity Coupé
- Sports Activity Vehicle

Number of seats

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 (66) from £ 23,550.00
5. BMW 3 Series Touring (64) from £ 24,865.00
6. BMW 3 Series Gran Turismo (39) from £ 29,200.00
Software product line (SPL) analysis

- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user

- Popular in embedded and critical systems domain: formal modelling and analysis techniques for proving SPL behaviour correct are widely studied

- Challenge existing formal methods and tools by potentially high number of different products, each giving rise to a large state space in general
As for CAS, scalability is an issue

Examples by C. Kästner (CMU, Pittsburgh, USA)

33 features

optional, independent

a unique configuration/variant for every

person on this planet
more configurations/variants than estimated atoms in the universe
Software product line (SPL) analysis

- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user.

- Popular in embedded and critical systems domain: formal modelling and analysis techniques for proving SPL behaviour correct are widely studied.

- Challenge existing formal methods and tools by potentially high number of different products, each giving rise to a large state space in general.

  ⇒ Lift success stories known for single systems (products) to sets of products (families) by exploiting variability modelling and analysis.
Product-based analysis:

- Simple, brute-force approach
- Make use of standardly available, highly-optimised analysis tools
Product-based analysis: inefficient, if not infeasible (example by S. Apel, University of Passau, Germany)

- Simple, brute-force approach
- Make use of standardly available, highly-optimised analysis tools

- Number of product variants is exponential in number of features
- Same piece of behaviour (or code) is verified numerous times, as many times as the number of variants that are able to execute it

---

**SQLite**

- **(88 features)**

---

**Time line**

- **Big bang**
- **Birth of Earth**
- **Now**
  - 1.37 * 10^{10} years
- **Measurement finished**
  - 2.9 * 10^{21} years
Solution: family-based analysis

- Beneficial in case of many products with substantial similarities
- Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it
Solution: family-based analysis

👍 Beneficial in case of many products with substantial similarities
👍 Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

👎 More complex analysis tasks
👎 Requires (compact) family models (superimposed, 150% models)
Solution: family-based analysis

👍 Beneficial in case of many products with substantial similarities

👍 Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

👎 More complex analysis tasks

👎 Requires (compact) family models (superimposed, 150% models)

💡 Dedicated variability-based models, logics, and model checkers (e.g. FTSs, Feature Nets, PL-CCS, fLTL, fCTL, QFLan, SNIP, VMC)

Dimovski et al., Family-based model checking without a family-based model checker. SPIN 2015
Chrszon et al., Family-based modeling and analysis for probabilistic systems – featuring ProFeat. FASE 2016
ter Beek et al., Family-Based Model Checking with mCRL2. FASE 2017
Dimovski et al., Variability-specific Abstraction Refinement for Family-based Model Checking. FASE 2017
Solution: family-based analysis

- Beneficial in case of many products with substantial similarities
- Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it
- More complex analysis tasks
- Requires (compact) family models (superimposed, 150% models)

Dedicated variability-based models, logics, and model checkers (e.g. FTSs, Feature Nets, PL-CCS, fLTL, fCTL, QFLan, SNIP, VMC)

- Dedicated model checkers need to be maintained and optimised
Solution: family-based analysis

- Beneficial in case of many products with substantial similarities
- Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

- More complex analysis tasks
- Requires (compact) family models (superimposed, *150% models*)

- Dedicated variability-based models, logics, and model checkers (e.g. FTSs, Feature Nets, PL-CCS, fLTL, fCTL, QFLan, SNIP, VMC)

- Dedicated model checkers need to be maintained and optimised

Dimovski et al., Family-based model checking without a family-based model checker. SPIN 2015

Chrszon et al., Family-based modeling and analysis for probabilistic systems – featuring ProFeat. FASE 2016

ter Beek et al., Family-Based Model Checking with mCRL2. FASE 2017

Dimovski et al., Variability-specific Abstraction Refinement for Family-based Model Checking. FASE 2017
Software product line (SPL) analysis

- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user.

- Popular in embedded and critical systems domain: formal modelling and analysis techniques for proving SPL behaviour correct are widely studied.


- Challenge existing formal methods and tools by potentially high number of different products, each giving rise to a large state space in general.

  ⇒ Lift success stories known for single systems (products) to sets of products (families) by exploiting variability modelling and analysis.

SPL vs. CAS: family-based model checking is a scalable verification technique (as are symbolic / mean field / statistical model checking) requiring to relate local (product) and global (family) system views.
Product line of (four) coffee machines with independent features \{\$, \€\}

\begin{tikzpicture}[node distance=2cm,auto,>=latex]
  
  
  
  \node[state] (s0) at (0,0) {$s_0$};
  \node[state] (s1) at (1,0) {$s_1$};
  \node[state] (s2) at (2,0) {$s_2$};

  
  
  
  \draw[->] (s0) edge [loop above] node {$F$} (s0);
  \draw[->] (s0) edge node {$std | T$} (s1);
  \draw[->] (s1) edge node {$ins | T$} (s0);
  \draw[->] (s1) edge node {$ins | \$\}$} (s2);
  \draw[->] (s2) edge node {$xxl | T$} (s1);

\end{tikzpicture}
Product line of (four) coffee machines with independent features \( \{ $, \euro \} \)
Product line of (four) coffee machines with independent features \{\$, €\}

Products with feature $\$\$ can obtain an xxl coffee upon coin insertion, but products without cannot.

how to express this? and how to model check this efficiently?
A feature $\mu$-calculus $\mu L_f$ over FTSs

Replace $\langle a \rangle \varphi$ and $[a] \varphi$ with $\langle a | \chi \rangle \varphi_f$ and $[a | \chi] \varphi_f$
Replace $\langle a \rangle \varphi$ and $[a] \varphi$ with $\langle a | \chi \rangle \varphi_f$ and $[a | \chi] \varphi_f$

$$F \quad \text{std}\mid T$$

$$\quad \text{ins}\mid T \quad \text{ins}\mid$${

$\quad s_0 \quad s_1 \quad s_2$

$$\quad \text{xxl}\mid T$$

$F | p_1 \quad \text{std}$$

$$\quad \text{ins} \quad \text{ins}$$

$$\quad s_0 \quad s_1 \quad s_2$$

$F | p_2 \quad \text{std}$$

$$\quad \text{ins}$$

$$\quad s_0 \quad s_1 \quad s_2$$

Products with feature $\$ can obtain an xxl coffee upon coin insertion, but products without cannot

$$[\text{true}^* | T] \left( ( [\text{ins} | \$] \langle \text{true}^* \cdot \text{xxl} | T \rangle \top ) \land [\text{xxl} | \neg \$] \bot \right)$$
A feature $\mu$-calculus $\mu L_f$ over FTSs

Replace $\langle a \rangle \varphi$ and $[a] \varphi$ with $\langle a|\chi \rangle \varphi_f$ and $[a|\chi] \varphi_f$

![Diagram of FTS with transitions](image)

Products with feature $\$$ can obtain an $xxl$ coffee upon coin insertion, but products without cannot

\[
\left[ \text{true}^* \mid T \right] \left( \left( \left[ \text{ins} \mid \$$ \right] \langle \text{true}^*, xxl \mid T \rangle T \right) \land \left[ xxl \mid \neg \$$ \right] \bot \right)
\]

Multi-feature $\mu L_f$ formula, more expressive than fLTL and fCTL
A first-order $\mu$-calculus with data $\mu L_{FO}$ over parametrised LTSs.
A first-order $\mu$-calculus with data $\mu L_{FO}$ over parametrised LTSs

$\mu L_{FO}$ is a fragment of the logic (e.g. $\langle a(v) \rangle$, $[a(v)]$) defined in:

Groote & Mateescu, Verification of temporal properties of processes in a setting with data. AMAST 1999

where its full semantics can be found; it is the logic used in
Main results

ter Beek et al., Family-Based Model Checking with mCRL2. FASE 2017
ter Beek et al., Towards a Feature mu-Calculus Targeting SPL Verification. FMSPLE 2016

From $\mu L_f$ to $\mu L_{FO}$ (and back)

Model checking a $\mu L_f$-formula over an FTS is equivalent to model checking a $\mu L_{FO}$-formula over the corresponding parametrised LTS.
From $\mu L_f$ to $\mu L_{FS}$ (and back)
Model checking a $\mu L_f$-formula over an FTS is equivalent to model checking a $\mu L_{FS}$-formula over the corresponding parametrised LTS

Family-based model checking $\mu L_f$
Satisfaction of $\varphi_f$ propagates from family to product level, but not vice versa (in general) and nothing can be said if $\varphi_f$ does not hold

⊕ if $\varphi_f$ holds for a family $P$, then for each individual product $p$, $\text{proj}(p, \varphi_f)$ holds too
**Main results**

ter Beek et al., Family-Based Model Checking with mCRL2. FASE 2017
ter Beek et al., Towards a Feature mu-Calculus Targeting SPL Verification. FMSPLE 2016

---

**From \( \mu L_f \) to \( \mu L_{FO} \) (and back)**
Model checking a \( \mu L_f \)-formula over an FTS is equivalent to model checking a \( \mu L_{FO} \)-formula over the corresponding parametrised LTS

**Family-based model checking \( \mu L_f \)**
Satisfaction of \( \varphi_f \) propagates from family to product level, but not vice versa (in general) and nothing can be said if \( \varphi_f \) does not hold

\( \oplus \) if \( \varphi_f \) holds for a family \( P \), then for each individual product \( p \),
\( \text{proj}(p, \varphi_f) \) holds too

\( \ominus \) if \( \varphi^c_f \) holds for a family \( P \), then for each individual product \( p \),
\( \text{proj}(p, \varphi_f) \) does not hold
Main results

From $\mu L_f$ to $\mu L_{FO}$ (and back)
Model checking a $\mu L_f$-formula over an FTS is equivalent to model checking a $\mu L_{FO}$-formula over the corresponding parametrised LTS

Family-based model checking $\mu L_f$
Satisfaction of $\varphi_f$ propagates from family to product level, but not vice versa (in general) and nothing can be said if $\varphi_f$ does not hold

- if $\varphi_f$ holds for a family $P$, then for each individual product $p$, $\text{proj}(p, \varphi_f)$ holds too
- if $\varphi_f^c$ holds for a family $P$, then for each individual product $p$, $\text{proj}(p, \varphi_f)$ does not hold

$\Rightarrow$ partition family $P$ into $(P_\oplus, P_\ominus)$, based on feature expressions
Family-based partitioning algorithm

$\text{FBP}(P, \varphi_f)$ terminates and returns a partitioning $(P_\oplus, P_\ominus)$ of $P$ s.t.

each product in $P_\oplus$ satisfies $\varphi_f$, while each product in $P_\ominus$ fails $\varphi_f$

**Algorithm 1** Family-Based Partitioning

```
1: function FBP($P, \varphi_f$)
2:     if $s_*, P \models_F \varphi_f$ then return $(P, \varnothing)$
3:     else
4:         if $s_*, P \models_F \varphi_c^c \varphi_f$ then return $(\varnothing, P)$
5:         else partition $P$ into $(P_1, P_2)$
6:             $(P_1^+, P_1^-) \leftarrow \text{FBP}(P_1, \varphi_f)$
7:             $(P_2^+, P_2^-) \leftarrow \text{FBP}(P_2, \varphi_f)$
8:         return $(P_1^+ \cup P_2^+, P_1^- \cup P_2^-)$
9:     end if
10: end if
11: end function
```
## Minepump SPL benchmark (2^7 products)


<table>
<thead>
<tr>
<th>( \Phi )</th>
<th>property</th>
<th>result</th>
<th>one-by-one</th>
<th>all-in-one</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi_1 )</td>
<td>Absence of deadlock</td>
<td>128/0</td>
<td>10.02</td>
<td>2.07</td>
</tr>
<tr>
<td>( \varphi_3 )</td>
<td>The controller cannot fairly receive each of the three message types</td>
<td>0/128</td>
<td>24.33</td>
<td>0.25</td>
</tr>
<tr>
<td>( \varphi_5 )</td>
<td>The system cannot be in a situation in which the pump runs indefinitely in the presence of methane</td>
<td>96/32</td>
<td>17.26</td>
<td>0.86</td>
</tr>
<tr>
<td>( \varphi_6 )</td>
<td>Assuming fairness (( \varphi_3 )), the system cannot be in a situation in which the pump runs indefinitely in the presence of methane (( \varphi_5 ))</td>
<td>112/16</td>
<td>27.32</td>
<td>3.67</td>
</tr>
<tr>
<td>( \varphi_7 )</td>
<td>The controller can always eventually receive/read a message, i.e. it can return to its initial state from any state</td>
<td>128/0</td>
<td>18.36</td>
<td>2.40</td>
</tr>
<tr>
<td>( \varphi_9 )</td>
<td>Invariantly, when the level of methane rises, it inevitably decreases</td>
<td>0/128</td>
<td>20.47</td>
<td>0.21</td>
</tr>
<tr>
<td>( \varphi_{11} )</td>
<td>Products with feature Ct can always switch on the pump</td>
<td>28/100</td>
<td>21.11</td>
<td>2.32</td>
</tr>
<tr>
<td>( \varphi_{12} )</td>
<td>Products with features Ct, Ma and Lh can start the pump upon a high water level, but products without feature Lh cannot</td>
<td>128/0</td>
<td>13.35</td>
<td>3.36</td>
</tr>
</tbody>
</table>

Clear improvement in run time when using mCRL2 for family-based as opposed to product-based model checking: average speed up ±31 (ranging from speed up of ±4 (\( \varphi_{12} \)) to a speed up of >97 (\( \varphi_3 \)))

---

mCRL2 code distributed with mCRL2 toolset (svn revision 14493)
The quest for an efficient strategy

Execution of Algorithm 1 for deadlock freedom ($\varphi_1$) and with initial family $\top$ (family characterised at node is conjunction of features along path from root)

Optional partitioning strategy

Total computation time: 27.9
Computation time leaves: 8.4
(i.e. $Mq$, $\neg Mq$, and $\neg Ct$ nodes)
At once $\forall$ possible families: 2.07

Non-optimal partitioning strategy
(splitting $Ln$ and $\neg Ln$, then optimal)
Total computation time: 45.0 (+60%)
Thanks!

Comments or Questions?

Contributors to Deliverable 3.3:
Maurice H. ter Beek (CNR), Luca Bortolussi (CNR), Vincenzo Ciancia (CNR), Stefania Gnesi (CNR), Jane Hillston (UEDIN), Diego Latella (CNR), Michele Loreti (IMT), Mieke Massink (CNR), Rytis Paškauskas (CNR), Mirco Tribastone (IMT), Max Tschaikowski (IMT) & Andrea Vandin (IMT)