Searching the Variability Space to Fix Model Inconsistencies: A Preliminary Assessment

Roberto E. Lopez-Herrejon, Alexander Egyed

Johannes Kepler University Linz, Austria

{roberto.lopez, alexander.egyed}@jku.at

Introduction

- ***Software Product Lines (SPL)***
  - Families of related systems distinguished by the features
  - Extensive success in achieving software reuse
  - *Variability*
    - The capacity of software artifacts to change
    - Its effective management is at the core of SPL

- ***Model-Driven Engineering (MDE)***
  - Emerging software development paradigm
  - Raises the level of abstraction and automates program generation

Motivation

- Fact
  - Increasing research convergence in SPL and MDE that leverages their complementary capabilities
- Challenge
  - Maintain consistency between models with variability
    - Checking that certain relations between model elements hold for ***all*** products of a SPL
- Problem
  - Research has focused on inconsistency detection
  - Inconsistency fixing has not been fully explored

Why Search-Based SE?

- Facts
  - SPL can employ multiple models simultaneously
    - State charts, sequence diagrams, class diagrams, …
  - Models can have a large number of consistency rules and instances
  - SPL usually involve large number of different products

- Because …
  - Large search space
  - No unique or optimal fixing solutions

Our ongoing work …

JOHANNES KEPLER UNIVERSITY LINZ | JKU

- Description
  - Finds fixing locations for single constraint instances
  - Uses basic search algorithm
- Relies on the notion of Safe Composition
  - Programming languages
    - guarantee that ***all*** programs of a product line are type safe
  - Uses propositional logic

Safe Composition

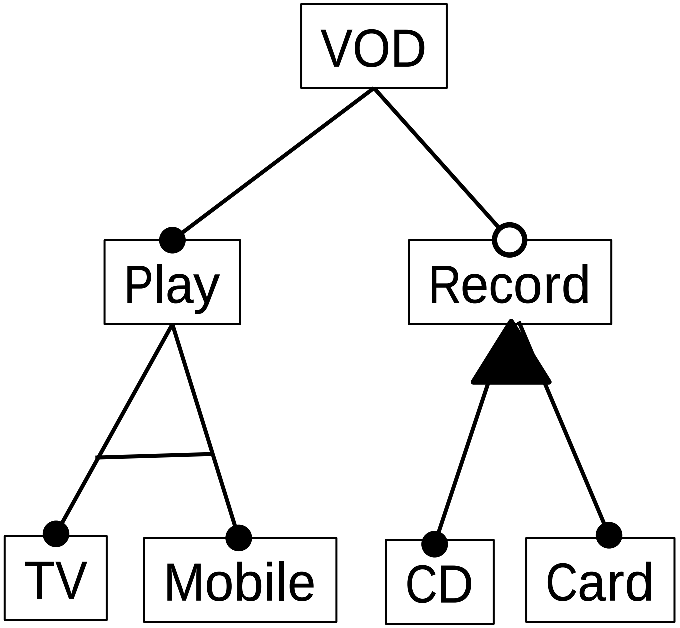
- Intuition
  - Implementation constraint(s) ($\texttt{IMP}_f$) must follow the product line domain constraints ($\texttt{PL}_f$)
- A SAT solver checks if ***one*** propositional formula is satisfiable or not
- Our interest is verifying that ***all*** the product line members satisfy an implementation constraint

$$\neg(\texttt{PL}_f \Rightarrow \texttt{IMP}_f)$$

Unsatisfiable = there is <u>no</u> product that violates the constraint

Satisfiable = there is <u>at least one</u> product that violates the constraint

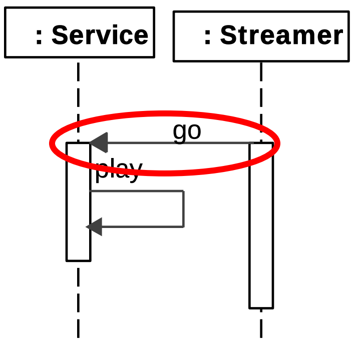
Example PL_f



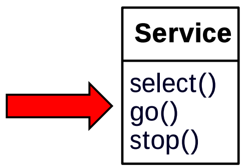
$$
\begin{aligned}
&\mathtt{VOD} \Leftrightarrow \mathtt{true} \quad \land \\
&\mathtt{VOD} \Leftrightarrow \mathtt{Play} \quad \land \\
&\mathtt{Record} \Rightarrow \mathtt{VOD} \quad \land \\
&\mathtt{TV} \Leftrightarrow \neg\mathtt{Mobile} \land \mathtt{Play} \quad \land \\
&\mathtt{Mobile} \Leftrightarrow \neg\mathtt{TV} \land \mathtt{Play} \quad \land \\
&\mathtt{Record} \Leftrightarrow \mathtt{CD} \lor \mathtt{Card}
\end{aligned}
$$

Requiring Rule Example

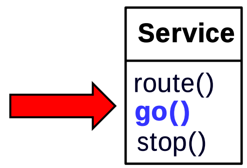
- Message action must be defined as an operation in receiver's class



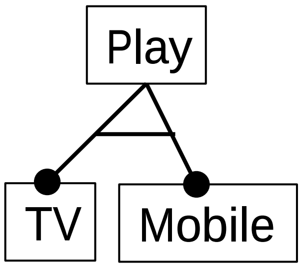
Feature Play



Feature TV

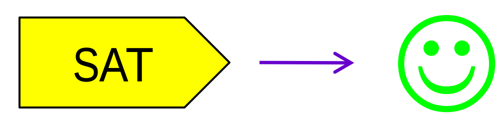


Feature Mobile



$$IMP_f \equiv Play \Rightarrow TV \vee Mobile$$

$$\neg(PL_f \Rightarrow IMP_f)$$



VOD ⇔ true ∧
VOD ⇔ Play ∧
Record ⇒ VOD ∧
TV ⇔ ¬Mobile ∧ Play∧
Mobile ⇔ ¬TV ∧ Play∧
Record ⇔ CD ∨ Card ∧

products with Mobile

Fixing inconsistencies

- Consistency Rule Instance (CRI) is a 4-tuple [F,RME,TS,FC] where:
  - Requiring feature F = Play
  - Requiring model element RME = $play_{msg}$
  - Set of pairs (feature, required elements) TS = {(TV, $play_{op}$)}
    - Set of features in the pairs of TS as TS[feature].
  - Faulty feature configuration that violates the consistency rule instance.
    FC = [{VOD, Play, Mobile},{TV, Record, CD, Card}]]

- Pair-wise commonality
  - Operation that receives a feature model P and two features F and G, and returns the number of products that have both features.

- Question: where to add the required elements?
  - Intuition: iteratively search fixing configurations choosing first those features with higher commonality → fix more configurations

Algorithm – Based on BFS

Input: CRI [F,RME,TS, FC] with FC$\neq \emptyset_{conf}$, and PL_f.

Output: Fixing set FS

Set of features that guarantees no faulty configurations

```
FC':=FC
FS:=TS[feature]
FSQ.enqueue(FS)
while FC' ≠ Ø_conf do
    FSQ.dequeue(FS)
    G:=maxCom(F,FC',TS,FS)
    FS:= FS U G
    FC':=SafeComposition(PL_f, F,FS)
    FSQ.enqueue(FS)
end while
return FS
```

Chooses a feature with maximum pair-wise commonality

Applies basic Safe Composition

Findings and The Road Ahead

- Preliminary evaluation
  - Using 60 publicly available feature models, 6-94 features
  - On average fixing sets around 5 elements
- Future work
  - Consider multiple consistency instances
  - Assess other search alternatives
  - Research alternatives for Pair-wise Commonality

Acknowledgements

MARIE CURIE ACTIONS

FWF Der Wissenschaftsfonds.

Questions?