IRISA

# Software Factories :
## *Software Product Lines with Model Driven Engineering*


***Prof. Jean-Marc Jézéquel***

**IRISA**

Campus de Beaulieu

F-35042 Rennes Cedex

Tel : +33 299 847 192 Fax : +33 299 847 171

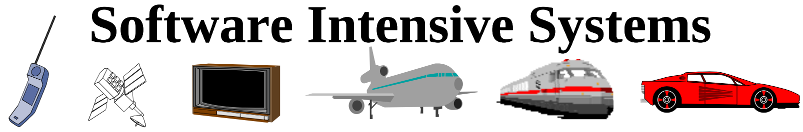
e-mail : jezequel@irisa.fr

http://www.irisa.fr/prive/jezequel

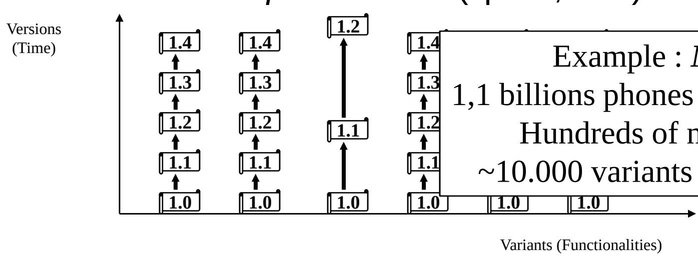

INRIA — Kermeta — UNIVERSITÉ DE RENNES 1

---

# Software Intensive Systems



- Importance of non-functional properties
  - distributed reactive systems, parallel & asynchronous
  - quality of service : security, reliability, latency, performance...
- Variations in functional aspects
  - notion of *product lines* (space, time)




Example : ***Nokia***

1,1 billions phones in circulation

Hundreds of networks

~10.000 variants of software

## Software Reuse

- drastically decrease cost of software development and maintenance
- increase quality of software
- reuse of existing software is one of the most promising approaches
- construct applications by composing reusable software pieces

(cost ⇨ TTM ⇨ staff)

3

## Two approaches to reuse

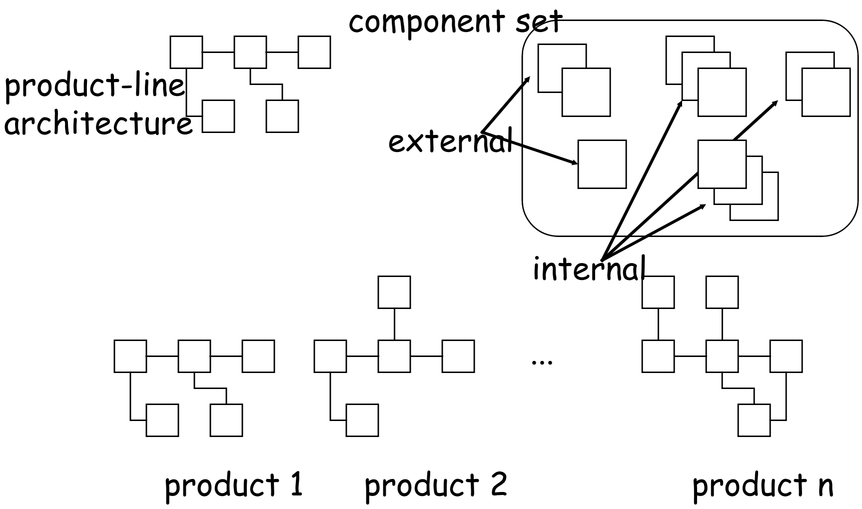
- Opportunistic:
  - the software engineer reuses pieces of software that fit the current problem and adds them to the software.
- Planned:
  - the organization puts explicit effort in developing reusable artifacts that provide the ‘right’ abstractions, ‘right’ level of variability and that fit into an higher level structure.
- Opportunistic reuse does not work in practice
  - As in automotive and other industries, build on the notion of ***Product Line***

4

# Software Product Lines

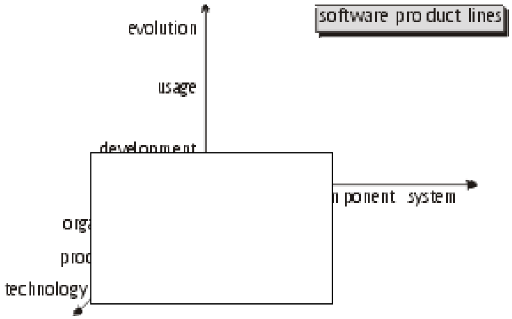
- A software product line consists of:
  - – product line architecture
  - – set of reusable components
  - – set of products, where each product has
    - » product architecture derived from PLA
    - » instantiated and configured components
    - » product-specific code

# Software Product Lines



## Many Issues Around SPLs

- Assets [Jacobsen]:
  - architecture
  - components
  - systems
- Views [SEI]:
  - business
  - organization
  - process
  - technology
- Lifecycle [Bosch]:
  - development
  - usage
  - evolution




11

# Technical Issues in SPLs: Managing variability

12

# Initiating a Product Line

| | Evolutionary | Revolutionary |
|---|---|---|
| Existing product line | Develop vision for PLA<br>Develop one comp. at a time by evolving existing components | Develop PLA, components and products based on requirement super set |
| New product line | PLA and components evolve based on requirements posed by new PL members | PLA and components developed to match requirements of all PL members |

13

# Evolving Existing Products

- advantages
  - reduced risk due to
    - small up front investment
    - early return on first shared components
  - relatively small effect on production schedule
- disadvantages
  - larger total investment

14

# Replace Existing Products

- advantages
  - shorter conversion time
  - smaller total investment
- disadvantages
  - higher risk
  - negative effect on production schedule

role of hardware and mechanical parts

15

# Applicability of SPL concepts

- context: consultancy company or IT department performing projects with partially overlapping requirements
- define common architecture and common components/subsystems
- develop (slightly) more general components in normal projects
- increase the variability and generality of the component in subsequent projects
- **balance investment and risk!**

16

## SPL Overview

- development
  - design product line architecture
  - develop software components
- deployment
  - develop members of product line
  - Ideally, "just" configure and specialize the product line
- evolution
  - evolve all assets in the product line
  - Software configuration management

17

## Model Driven Engineering for SPL

- How to model PL with the UML?
  - Variability in the UML class diagrams
- Constraints on PL models
  - The use of OCL (Object Constraints Language ) to specify PL constraints
- The PL derivation
  - The decision model
  - The PL derivation as the UML models transformation

18

# The 3 Dimensions of Software Configuration Management:

**[Estublier et al. 95]**

- The Variant dimension
  – Handle environmental differences
- The Revision dimension
  – Evolution over time
- The Concurrent Activities dimension
  – Many developers are authorized to modify the same configuration item
- Even with the help of sophisticated tools, the complexity might be daunting
  – Try to simplify it by reifying the variants of an OO system

# Variants in Software Systems

- Hardware Level
- Heterogeneous Distributed Systems
- Peculiarities in Target Operating System
- Compiler Differences
- Range of Products
- User Preferences for GUI
- Internationalization

- Vi= 16
- Vp = 4
- Vn = 8
- Vg = 5
- Vl = 24

- Number of Variants = $Vp*Vn*Vg*2^{Vi+Vl-2}$
  – That's 43,980,465,111,040 possible variants

# Traditional Approaches

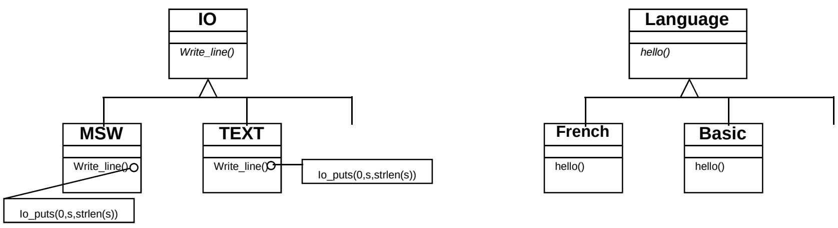
- Patch the executable
- Device Drivers
  - source level, link time, boot time, on demand at runtime
- Static Configuration Table
- Conditional Compilation / Runtime Tests

```
If (language == french) {
#ifdef MSW
            io_puts(0,"Bonjour",7);
#elsif TEXT
            printf("Bonjour\n");
#endif
} else {
#ifdef MSW
            io_puts(0,"Hello",5);}
#elsif TEXT
            printf("Hello\n");}
#endif
```

- Static and Dynamic configuration information intermingled
- Hard to change your mind on what should be static or dynamic...

# Basic Idea

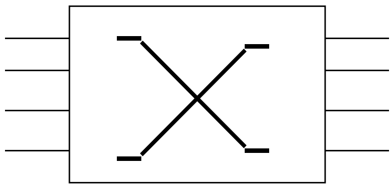
- Abstract the Intent
  - io.write_line(language.hello)
- Rely on Dynamic Binding for the Details
  - Don't care now for static/dynamic distinction



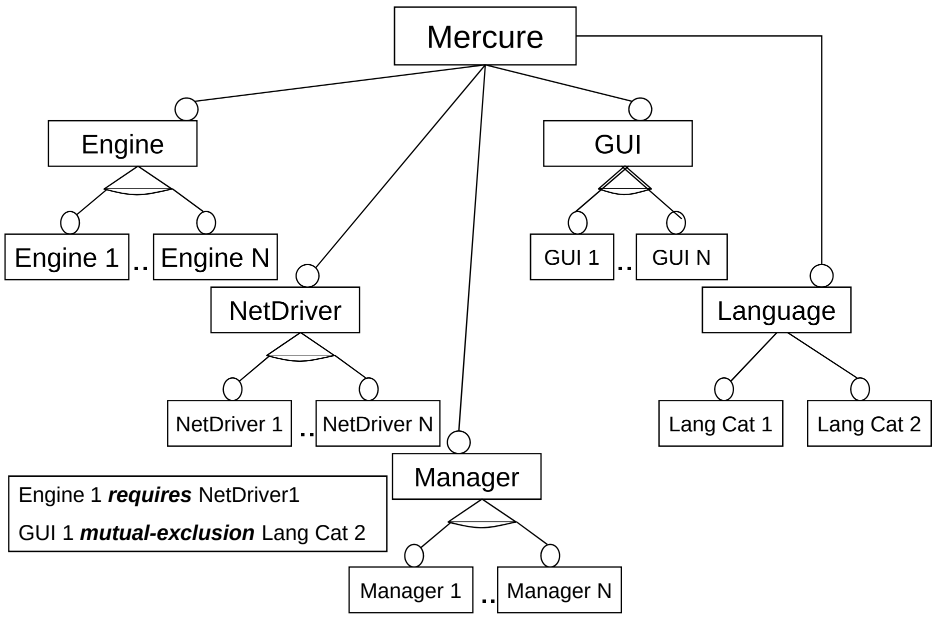

- Uncouple the variations from the selection process
  - Automatically derive a product using OCL 2 meta-model transformation

# The Mercure PL as example

- A family of SMDS[Jezequel 96] (Switched Multi-Megabits Data Service) servers.
- Delivering, forwarding, and relaying messages from and to a set of network interfaces.



23




24

# Variability in UML class diagrams

Abstraction

Inheritance, Abstract Factory

- Parameterization

UML class templates

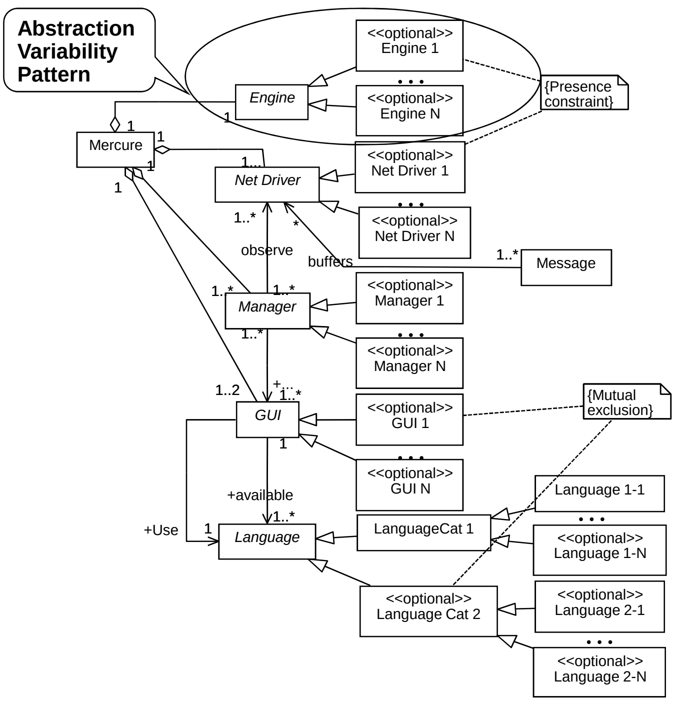
- Optionality

UML extensions mechanisms (Stereotype « Optional »)

- Alternatives

Optional + Constraint

25




27

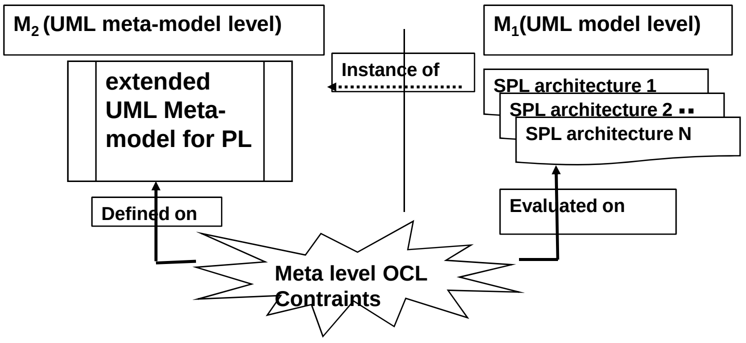
## Product Line Constraints

- Generic Contraints (GC): Constraints for ALL product line architectures (architecture coherence)

- Specific Constraints (SC): Constraints for a SPECIFIC product line architecture (relationships between specific elements).

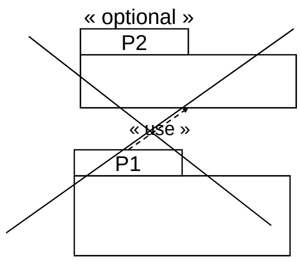
28

## GC as OCL meta-level constraints




29

# Examples



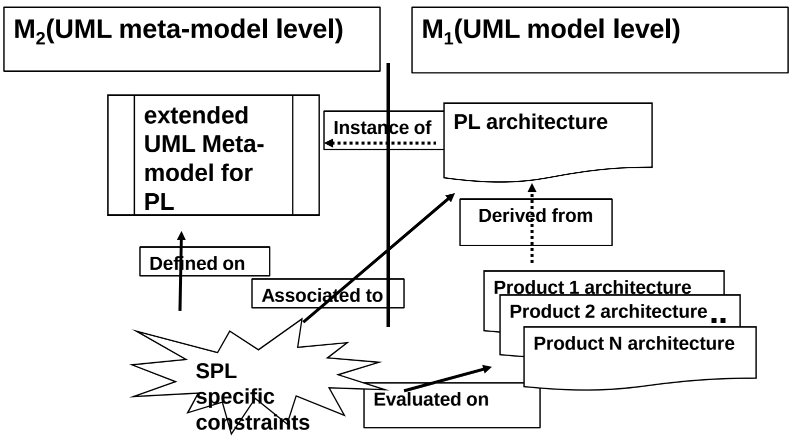

## Dependency constraint

context **Dependency**

inv : **self.supplier** → exists**(S:ModelElement S.isStereotyped('optional')**
implies **self.client** → forAll**( C:ModelElement | C.isStereotyped('optional') )**

30

# SC as OCL meta-level constraints




31

# Examples

- Presence constraint

  context **Model_Management::Model**

  inv:

  **self.presenceClass('ENGINE1')** implies
  **self.presenceClass('NETDRIVER1')**

- Mutual exclusion constraint

  context **Model_Management::Model**

  inv:

  **(self.presenceClass('GUI1')** implies not
  **(self.presenceClass('LANGUAGE_CAT2'))** and
  **(self.presenceClass('LANGUAGE_CAT2')** implies not
  **self.presenceClass('GUI1'))**

32

# Product Line Derivation

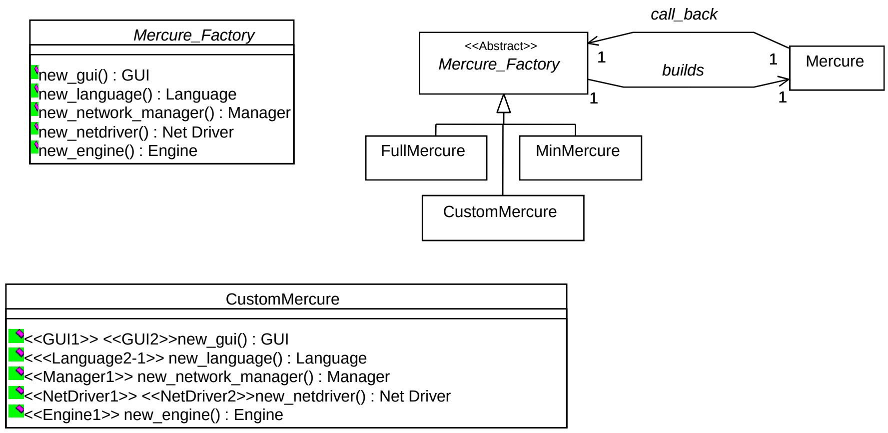
| Products \ VPs | GUI | Language | Manager | NetDriver | Engine |
|---|---|---|---|---|---|
| CustomMercure | GUI1, GUI2 | Language 2-1 | Manager1 | NetDriver1 | Engine1 |
| MiniMercure | GUI1 | Language 1 | Manager1 | NetDriver1 | Engine1 |
| FullMercure | all | all | all | all | all |

GUI 1 ***mutual-exclusion*** Lang Cat 2

33

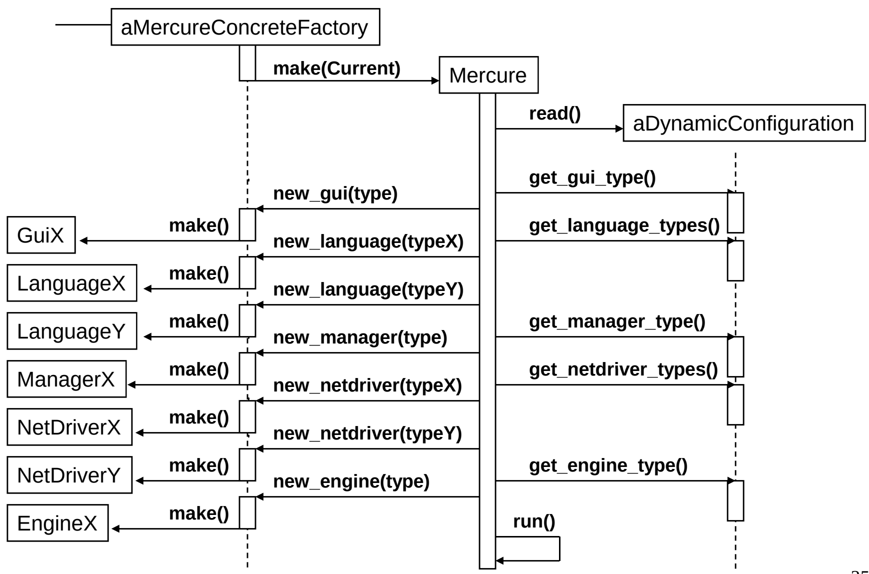
# The decision model

- The Abstract Factory Design Pattern
  - [Gamma et al 95]




34

# Dynamic Configuration




35

# The PL derivation

- Manipulate the UML meta-model to automatically derive a product model using a Model Transformation Language (MTL)

<u>DeriveProductLine</u>

***Input*:** **PL_model:** Model
**aConcreteFactory:** Class
pre **: -- check Generic Constraints on PL_model**
***Output* :** **Product_model:** Model
post **:-- check Specific Constraints on the PL_model**

36

# Model Transformation

- By limiting the range of variants available from a given Concrete Factory:
  - The transformer may know the set of *living* classes
    - » special case of Partial Evaluation
  - Generate a specialized model for the product
    - » When only one *living* class for an abstract varying part:
      - Dynamic binding replaced with direct call (and even inlining)
    - » When only a few *living* classes
      - Dynamic binding replaced by *if then … else*
      - Implemented in e.g., GNU SmallEiffel
- All static configuration issues kept encapsulated and do not pollute the model

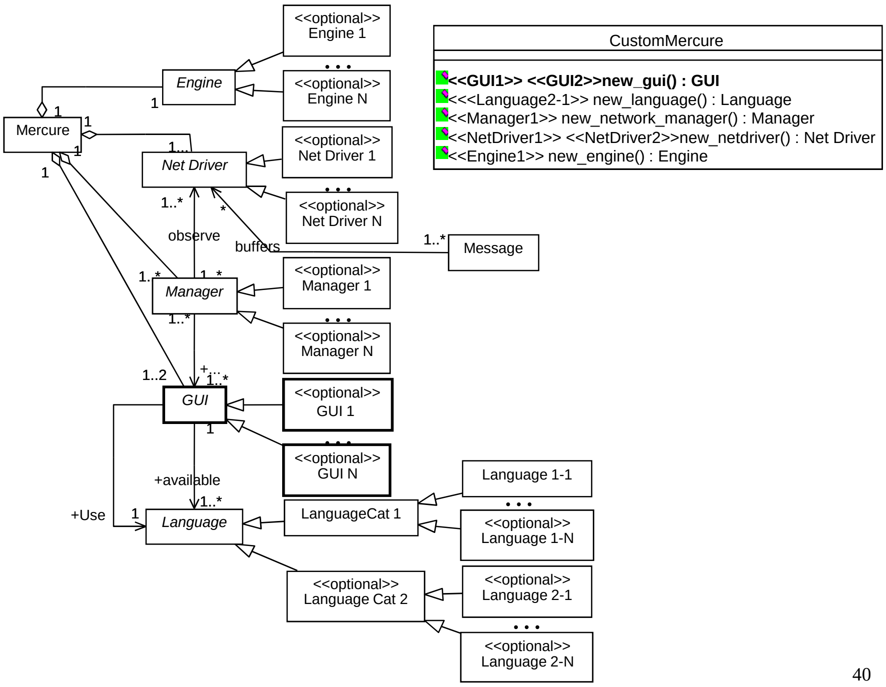
# The PL derivation

- The Variants selection:
  - Using operation factory stereotypes
- The Model specialization:
  - Removes all optional classes which have not been selected
- The Model optimization:
  - Deletes unused factories, Optimize inheritance

38

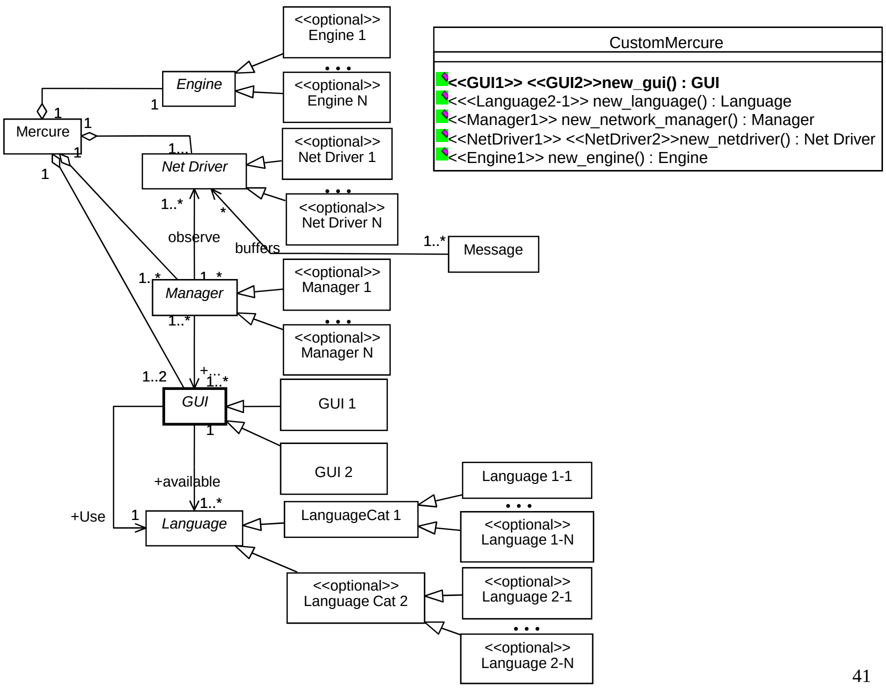
```
DerivePL (PL_model: Model, aConcreteFactory: Class) : Model  {

    selectedVariants: Set;
    Result := clone(PL_model);
    selectedVariants := getSelectedVariants(aConcreteFactory);
    // Model specialization
   for each optional class C in PL_model do
        if     (the class name of C not in selectedVariants)
           and  names of all subclasses of C not in selectedVariants)
       then
           delete the class C from Result;
       endif
   done
// Model optimization
     replace abstract classes with only one subclass S by S
    delete all other factories
}
```

39



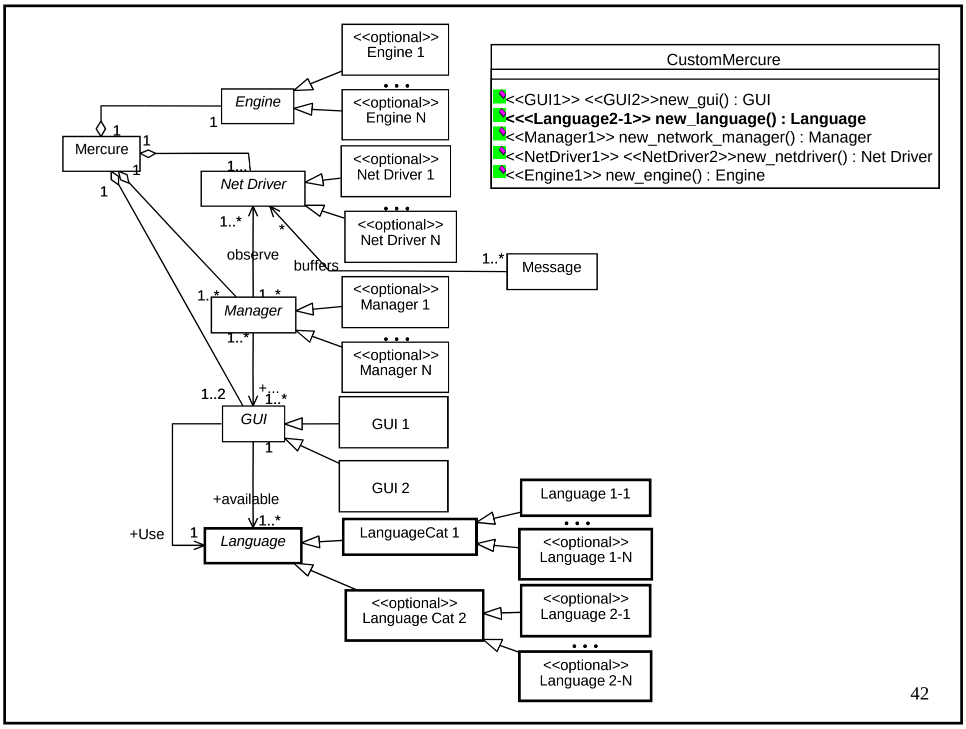
<<optional>> Engine 1
Engine
<<optional>> Engine N
Mercure
<<optional>> Net Driver 1
Net Driver
<<optional>> Net Driver N
observe
buffers
Message
Manager
<<optional>> Manager 1
<<optional>> Manager N
GUI
<<optional>> GUI 1
<<optional>> GUI N
+available
+Use
Language
LanguageCat 1
Language 1-1
<<optional>> Language 1-N
<<optional>> Language Cat 2
<<optional>> Language 2-1
<<optional>> Language 2-N
CustomMercure
<<GUI1>> <<GUI2>>new_gui() : GUI
<<<Language2-1>> new_language() : Language
<<Manager1>> new_network_manager() : Manager
<<NetDriver1>> <<NetDriver2>>new_netdriver() : Net Driver
<<Engine1>> new_engine() : Engine


40

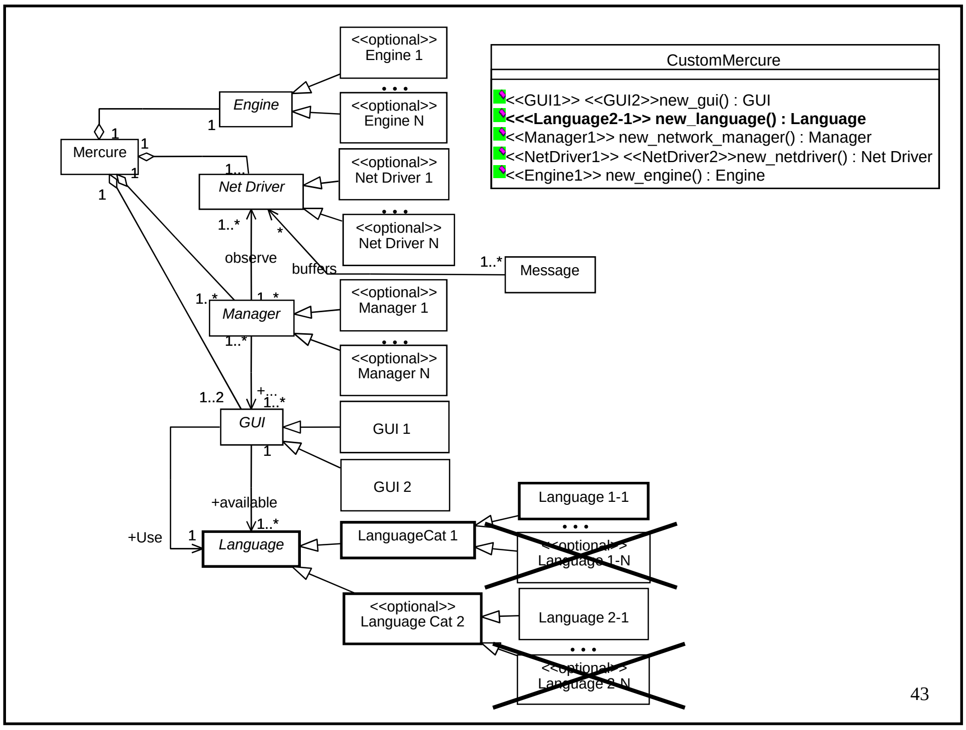


<<optional>> Engine 1
Engine
<<optional>> Engine N
Mercure
<<optional>> Net Driver 1
Net Driver
<<optional>> Net Driver N
observe
buffers
Message
Manager
<<optional>> Manager 1
<<optional>> Manager N
GUI
GUI 1
GUI 2
+available
+Use
Language
LanguageCat 1
Language 1-1
<<optional>> Language 1-N
<<optional>> Language Cat 2
<<optional>> Language 2-1
<<optional>> Language 2-N
CustomMercure
<<GUI1>> <<GUI2>>new_gui() : GUI
<<<Language2-1>> new_language() : Language
<<Manager1>> new_network_manager() : Manager
<<NetDriver1>> <<NetDriver2>>new_netdriver() : Net Driver
<<Engine1>> new_engine() : Engine

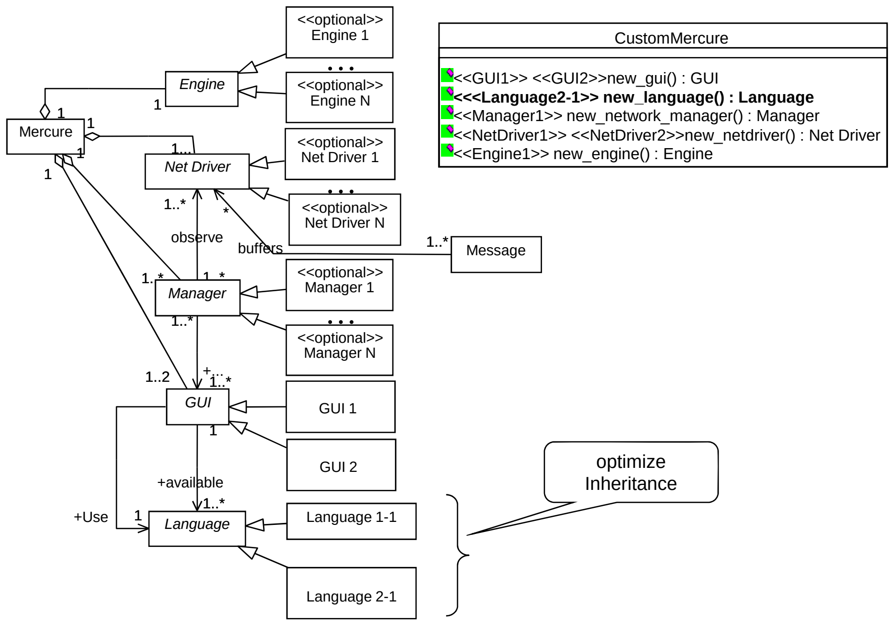

41



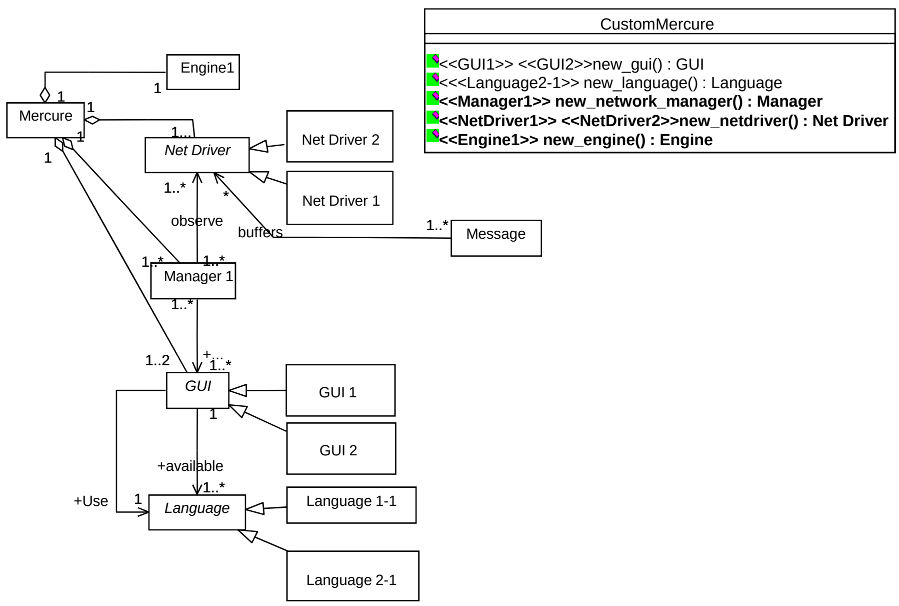
<<optional>> Engine 1
Engine
<<optional>> Engine N
Mercure
<<optional>> Net Driver 1
Net Driver
<<optional>> Net Driver N
observe
buffers
Message
Manager
<<optional>> Manager 1
<<optional>> Manager N
GUI
GUI 1
GUI 2
+available
+Use
Language
LanguageCat 1
Language 1-1
<<optional>> Language 1-N
<<optional>> Language Cat 2
<<optional>> Language 2-1
<<optional>> Language 2-N
CustomMercure
<<GUI1>> <<GUI2>>new_gui() : GUI
<<<Language2-1>> new_language() : Language
<<Manager1>> new_network_manager() : Manager
<<NetDriver1>> <<NetDriver2>>new_netdriver() : Net Driver
<<Engine1>> new_engine() : Engine
42

<<optional>> Engine 1
Engine
<<optional>> Engine N
Mercure
<<optional>> Net Driver 1
Net Driver
<<optional>> Net Driver N
observe
buffers
Message
Manager
<<optional>> Manager 1
<<optional>> Manager N
GUI
GUI 1
GUI 2
+available
+Use
Language
LanguageCat 1
Language 1-1
<<optional>> Language 1-N
<<optional>> Language Cat 2
Language 2-1
<<optional>> Language 2-N
CustomMercure
<<GUI1>> <<GUI2>>new_gui() : GUI
<<<Language2-1>> new_language() : Language
<<Manager1>> new_network_manager() : Manager
<<NetDriver1>> <<NetDriver2>>new_netdriver() : Net Driver
<<Engine1>> new_engine() : Engine
43

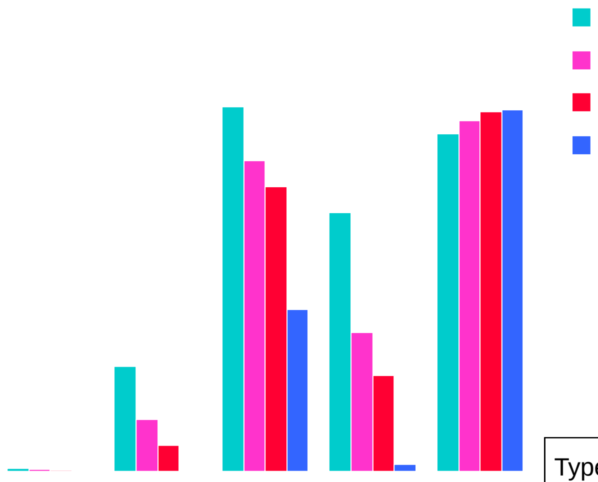

CustomMercure
<<GUI1>> <<GUI2>>new_gui() : GUI
**<<<Language2-1>> new_language() : Language**
<<Manager1>> new_network_manager() : Manager
<<NetDriver1>> <<NetDriver2>>new_netdriver() : Net Driver
<<Engine1>> new_engine() : Engine
<<optional>> Engine 1
<<optional>> Engine N
Engine
Mercure
Net Driver
<<optional>> Net Driver 1
<<optional>> Net Driver N
observe
buffers
Message
Manager
<<optional>> Manager 1
<<optional>> Manager N
GUI
GUI 1
GUI 2
+available
+Use
Language
Language 1-1
Language 2-1
optimize Inheritance

44


CustomMercure
<<GUI1>> <<GUI2>>new_gui() : GUI
<<<Language2-1>> new_language() : Language
**<<Manager1>> new_network_manager() : Manager**
**<<NetDriver1>> <<NetDriver2>>new_netdriver() : Net Driver**
**<<Engine1>> new_engine() : Engine**
Engine1
Mercure
Net Driver
Net Driver 2
Net Driver 1
observe
buffers
Message
Manager 1
GUI
GUI 1
GUI 2
+available
+Use
Language
Language 1-1
Language 2-1

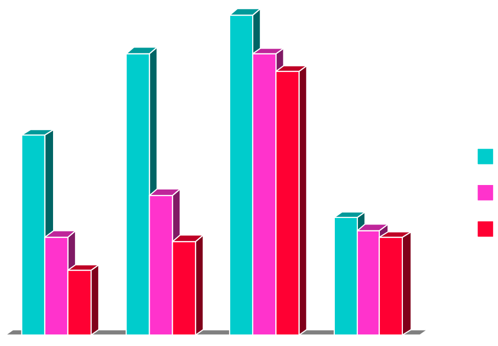
45

# Code size



Type inference score for MiniMercure is 99.7%

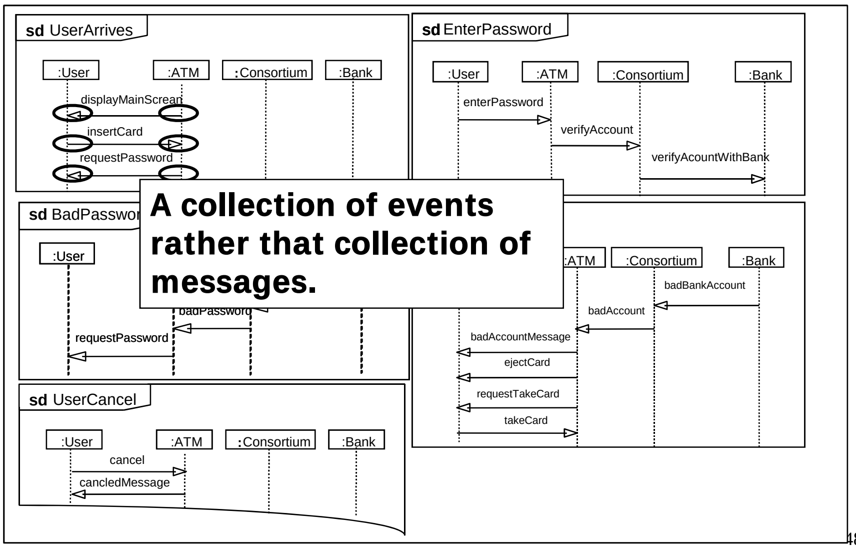
# Runtime Performances



Dead code removal

Automatic static binding

## Derivation of products from product-lines: behavioral issues based on Sequence Diagrams

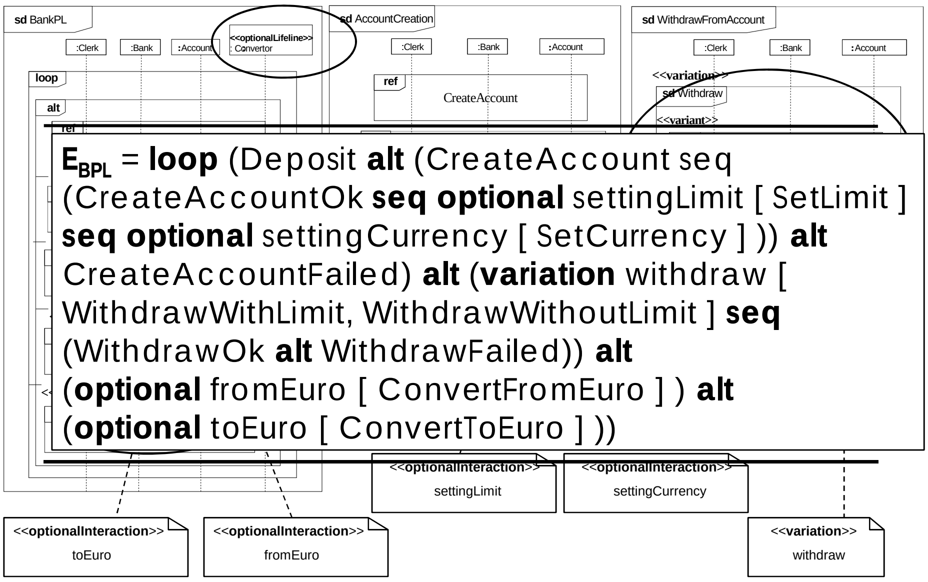



48

## HMSCs & UML2.0 SDs: Combined SDs

- A combined SD: refers to a set of interactions and composes them by means of operators:
  - **Sequence** (**seq**): weak sequential composition.
  - **Alternative** (**alt**): choice between interaction operands.
  - **Loop** (**loop**): iteration of an interaction.
- Extended with operators to model variability
  - **Optional, variation, virtual**…

49




$E_{BPL}$ = **loop** (Deposit **alt** (CreateAccount seq (CreateAccountOk **seq optional** settingLimit [ SetLimit ] **seq optional** settingCurrency [ SetCurrency ] )) **alt** CreateAccountFailed) **alt** (**variation** withdraw [ WithdrawWithLimit, WithdrawWithoutLimit ] **seq** (WithdrawOk **alt** WithdrawFailed)) **alt** (**optional** fromEuro [ ConvertFromEuro ] ) **alt** (**optional** toEuro [ ConvertToEuro ] ))

50

# STEP 1: Behavioral derivation

| Product | Decision model instance (DMI) |
|---|---|
| BS1 | **DM1** ={(settingLimit, TRUE), (settingCurrency, FALSE), (withdrawAccount, 1), (fromEuro, FALSE), (toEuro, FALSE)} |
| BS2 | **DM2** ={(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, FALSE), (toEuro, FALSE)} |
| BS3 | **DM3** ={(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, TRUE), (toEuro, TRUE)} |
| BS4 | **DM4** ={(settingLimit, TRUE), (settingCurrency, TRUE), (withdrawAccount, 1), (fromEuro, TRUE), (toEuro, TRUE)\} |

**RESD = [[PL-RESD]]** $_{\text{DMi}}$

51

# Behavioral derivation rules

- [[**optional** name [ **E** ] ]] $_{DMi}$ = { E **IF** *(name, TRUE)* $\in$ *DMi* ; $E_\varnothing$ **IF** *(name, FALSE)* $\in$ *DMi* }

$E_\varnothing$ is the *empty* SD: neutral element for **seq**, **alt** ; idempotent for **loop**

- [[**variation** name [ **E1, E2,..** ] ]] $_{DMi}$ = Ei **IF** *(name, i)* $\in$ *DMi*

- [[**virtual** name [ **E** ] ]] $_{DMi}$ = Ereff **IF** *(name, Ereff)* $\in$ *DMi, E* **ELSE**

52

**DM2** ={(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, FALSE), (toEuro, FALSE)}

$E_{BS2}$ = [[ $E_{BPL}$]] $_{DM2}$

**$E_{BS2}$** = **loop** (Deposit **alt** (CreateAccount seq
(CreateAccountOk **seq** $E_\varnothing$
**seq** $E_\varnothing$ )) **alt**
CreateAccountFailed) **alt** ( **WithdrawWithoutLimit seq**
(WithdrawOk **alt** WithdrawFailed)) **alt**
( $E_\varnothing$ ) **alt**
( $E_\varnothing$ ) )

- $E_\varnothing$ is the *empty* SD: neutral element for **seq**, **alt** ; idempotent for **loop**
  - ➔ **expression reduction**

53

**DM2** ={(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, FALSE), (toEuro, FALSE)}

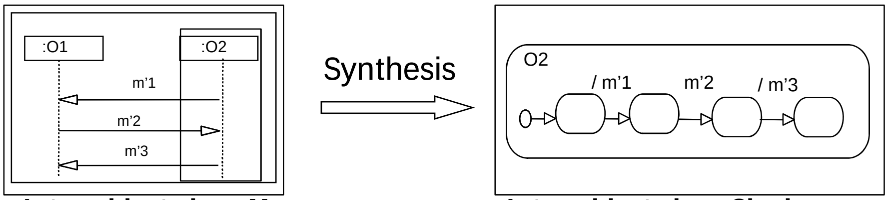
$E_{BS2}$ = [[ $\mathbf{E_{BPL}}$]] $_{DM2}$

$\mathbf{E_{BS2}}$ = **loop** (Deposit **alt** (CreateAccount seq (CreateAccountOk )) **alt** CreateAccountFailed) **alt** ( **WithdrawWithoutLimit seq** (WithdrawOk **alt** WithdrawFailed)))

<u>Step 1 result</u> : One expression (RESD) for each product

54

# STEP2: UML Sequence Diagrams (SDs): ➔ Statecharts



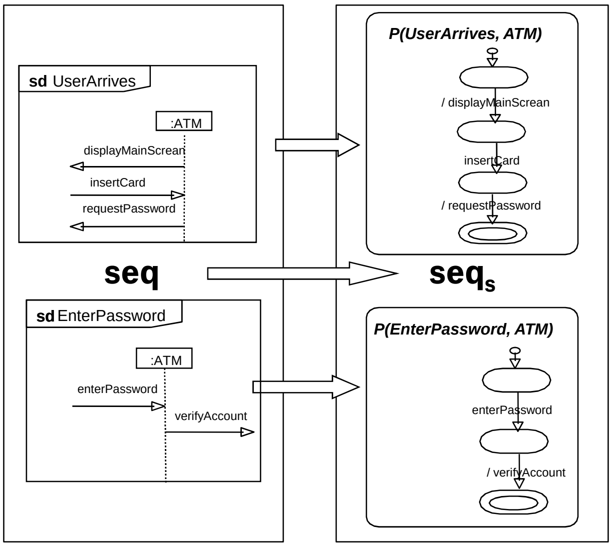

**Inter-object view: Many objects, <u>one example</u>.**

**Intra-object view: Single object, <u>a complete behavior</u>.**

- Related work:
  - *Kriss et al* [UML 98],
  - *Koskimies et al [IEEE Software 98]*
  - *Whittle et al* [ICSE 00] ,
  - *Mäkinen et al* [ICSE 01]..etc

55

# From combined SDs

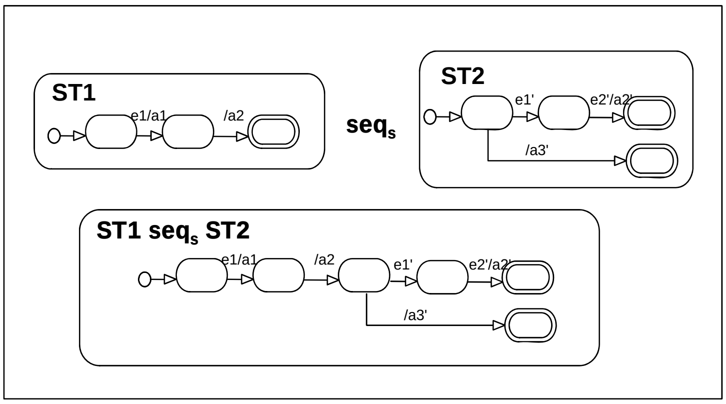



| SDs | STs |
|---|---|
| **seq** | **seq$_s$** |
| **alt** | **alt$_s$** |
| **loop** | **loop$_s$** |

56

# Statecharts operators

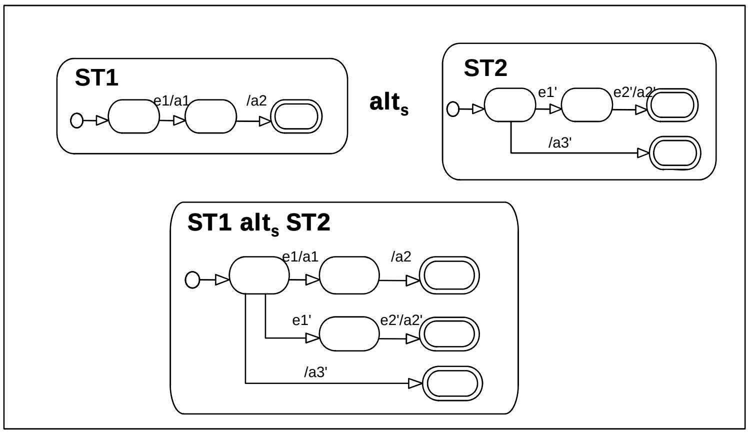
- 1) Sequence (**seq$_s$**)




57

# Statecharts operators(contd.)

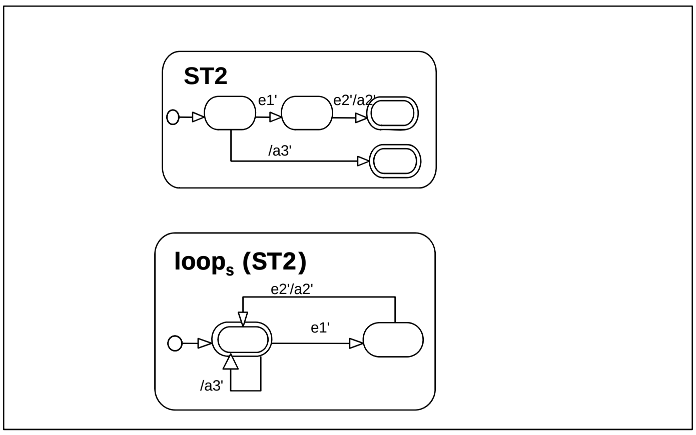
– 2) Alternative (**alt$_s$**)




58

# Statecharts operators(contd.)

– 3) Iteration (**loop$_s$**)




59

## From combined SDs (contd.)

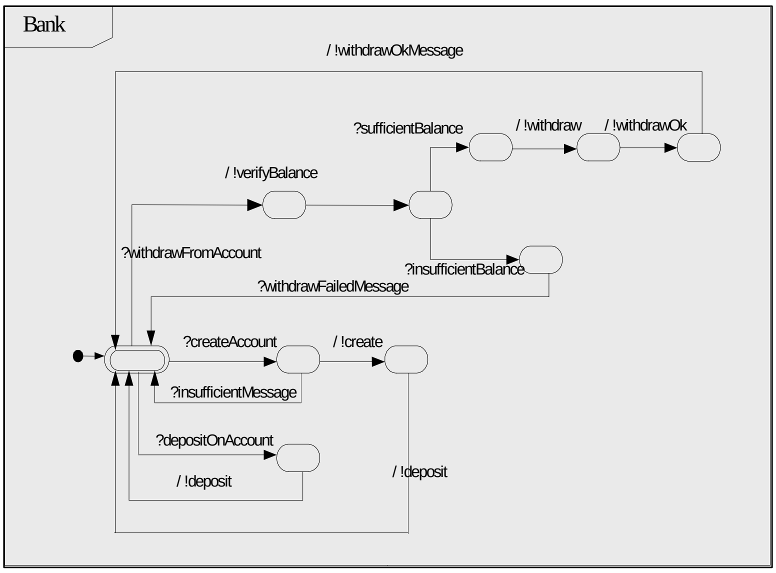
**E** = **loop** (UserArrives **seq** ( **loop** (EnterPassword **seq** BadPassword) **seq** EnterPassword **seq** (BadAccount **alt** UserCancel) **alt** UserCancel))

⇩

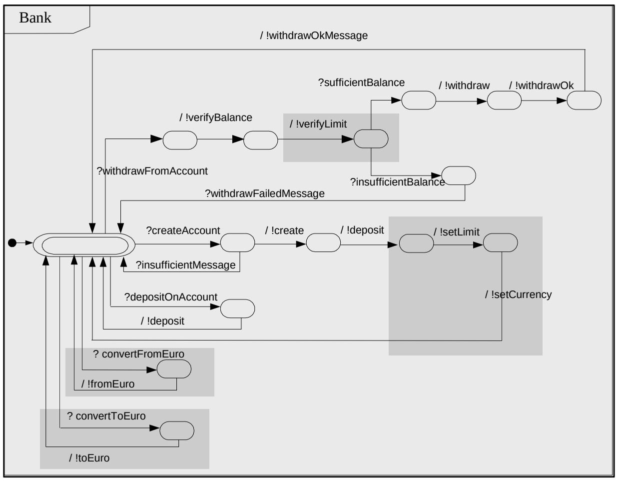
**E** = **loop**$_s$ (P(UserArrives, ATM) **seq**$_s$ ( **loop**$_s$ (P(EnterPassword,ATM) **seq**$_s$ P(BadPassword, ATM)) **seq**$_s$ P(EnterPassword,ATM) **seq**$_s$ (P(BadAccount,ATM) **alt**$_s$ P(UserCancel,ATM)) **alt**$_s$ P(UserCancel,ATM)))



## Result: Bank StateChart for BS2



## Result: Bank StateChart for BS4




62

# Conclusion & Perspectives

63

## Conclusion

- Handling variability in SPL is complex
  - Exponential number of configurations
- Use models (& aspects) to manage complexity
  - Abstract away from #IFDEF & diff
- Executable Meta-Modeling to Automate Product Derivation from SPL Models
  - Both from Static & Dynamic aspects

64

## Perspectives on SPL Research

- Composition of models
  - New ways of composing software from modeling elements
    - » at both model and meta-model levels
    - » Unifying MDE, AOSD, SPL, Generative Programming…
  - Composing models at runtime: dynamic adaptation
    - » FP7 STREP: DiVA (03/2008-03-2011)

65