



A Trial on Model Based Test Case Extraction and Test Data Generation

Xiaojing ZHANG, Takashi HOSHINO

NTT Cyber Space Laboratories
Tokyo, JAPAN

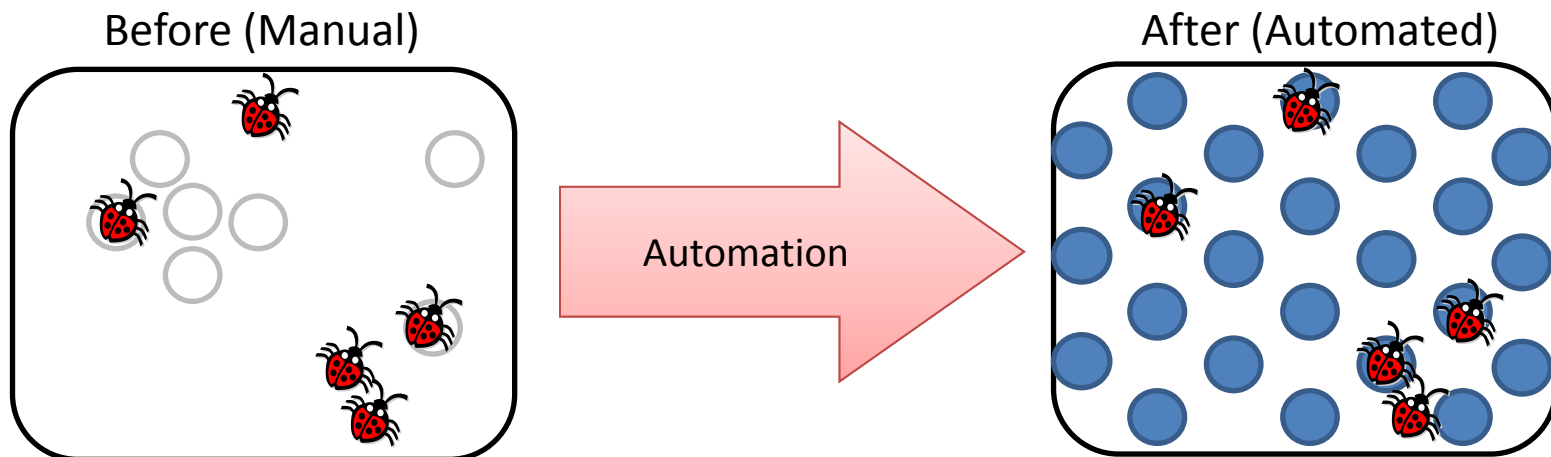
- Background and motivation
- Generation approach
- Tool implementation
- Evaluation results
- Future work and conclusion

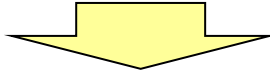
- We need better quality assurance!
 - Software defects become a public concern.
- Why testing?
 - The last check on the final product before release.
- What's testing?
 - A confirmation of whether the product is developed just as one intended.
- How to improve testing?
 - Quantity
 - $\text{Test density} = \text{Number of test cases} / \text{Size of the SUT.}$
 - Quality
 - $\text{Structure coverage} = \text{Elements tested} / \text{Total number of elements.}$
 - $\text{Input space coverage} = \text{Inputs used for testing} / \text{Entire input space.}$

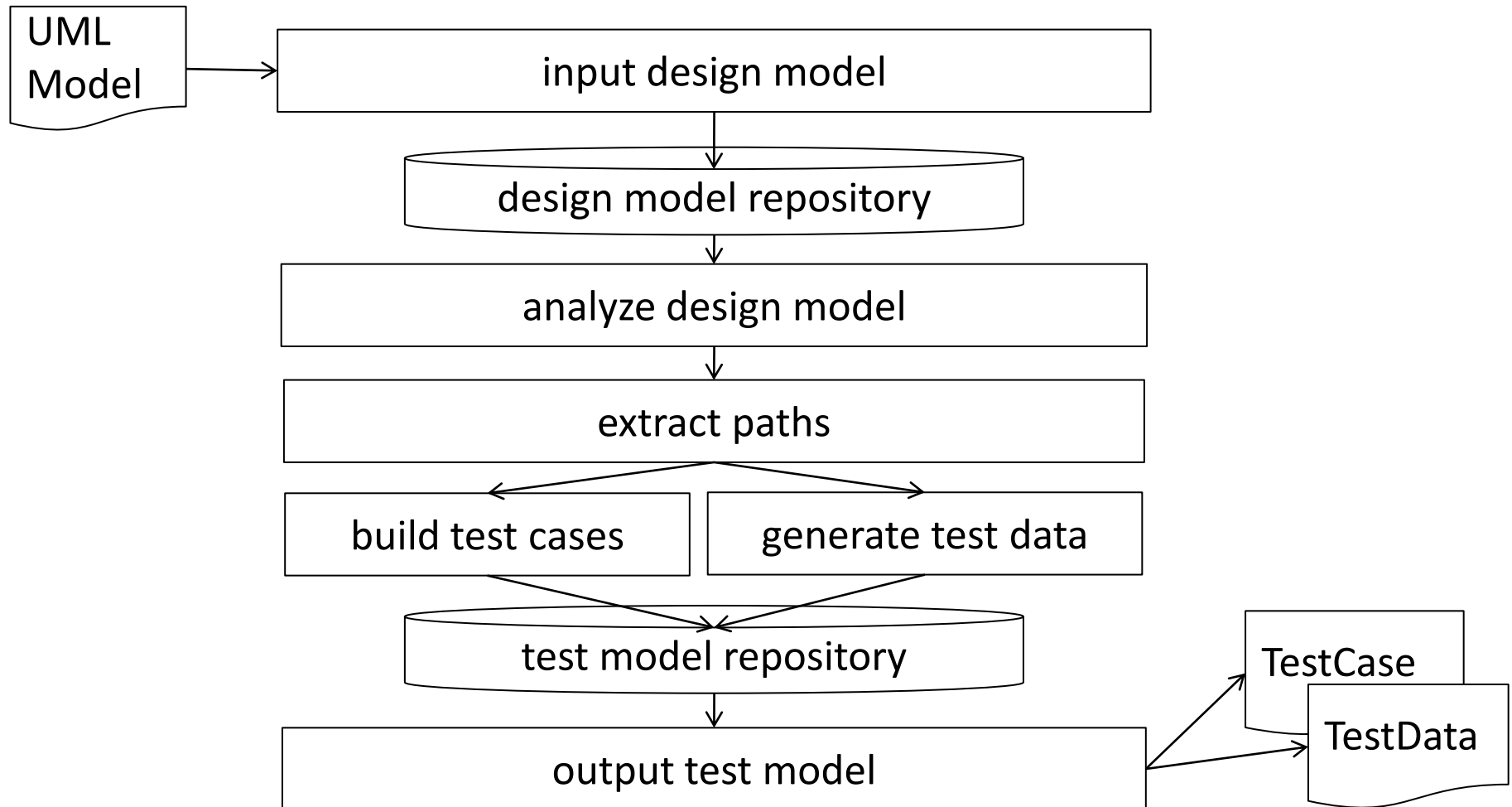
- **our goal is to...
improve “test design”, with low cost by automation**

– Test design:

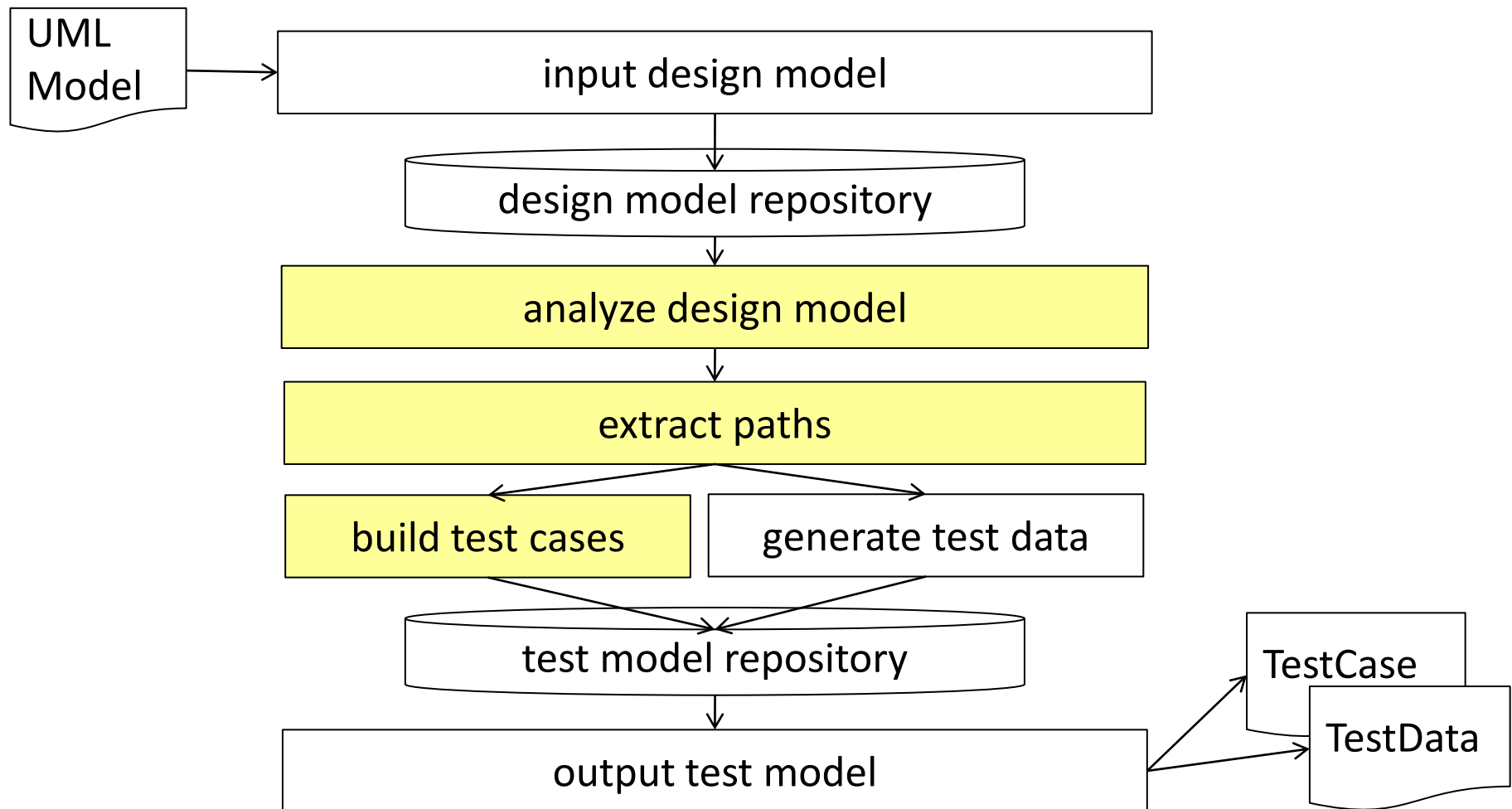
- Extracting test cases and test data for test execution
- Needs huge effort to achieve high density and coverage



- Many literatures on
 - Generation from source code or original design notation.
 - Primitive data type like integer or string
- 
- easy notation
 - UML 2.0 activity for behavior, class for data structure
 - design-based generation
 - Suppose that the intentions of the customer are formalized into software design
 - test data with structure
 - More variations have to be considered when the test data has hierarchical or repetition structure



Test Case Generation



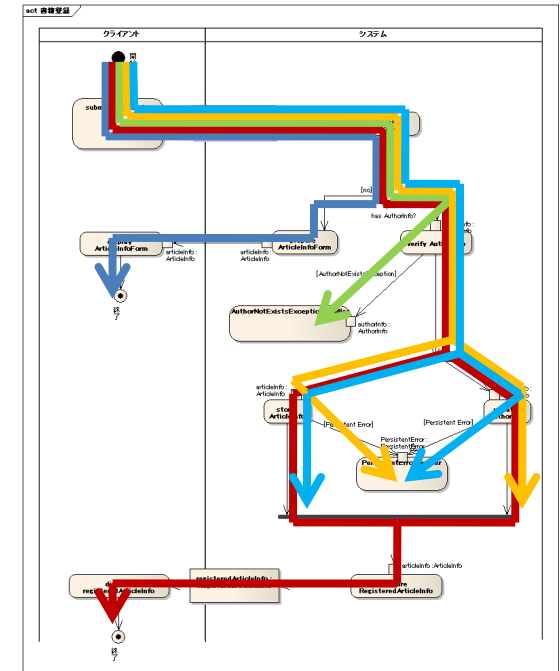
1. Decompose UML model into activities

2. Extract “paths” from UML Activity

- simple depth first search algorithm
- If a loop exists,
case of not passing the loop
case of passing the loop just once

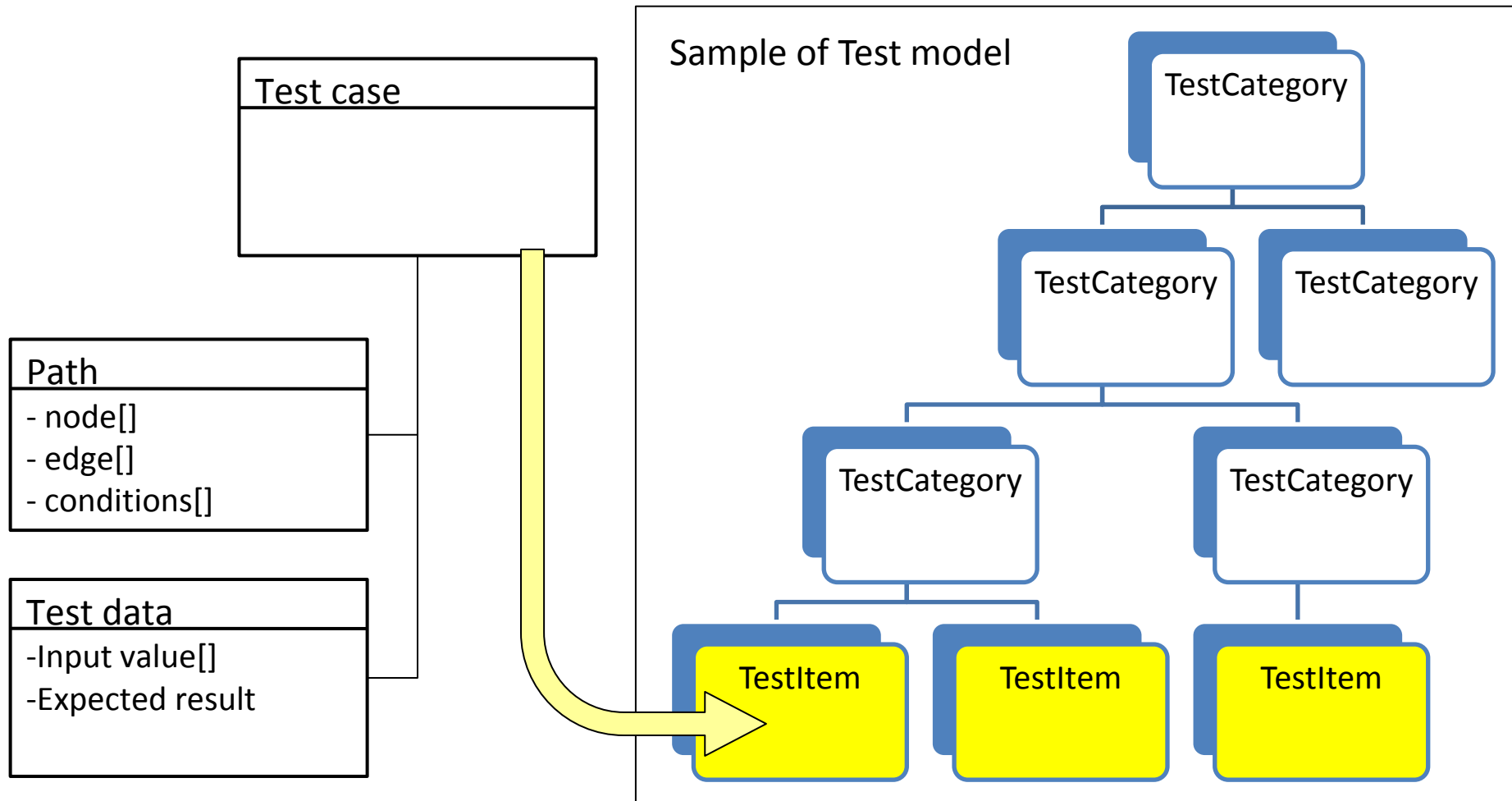
3. Make one test case for each path

- a test case is a pair of a particular path and the test data

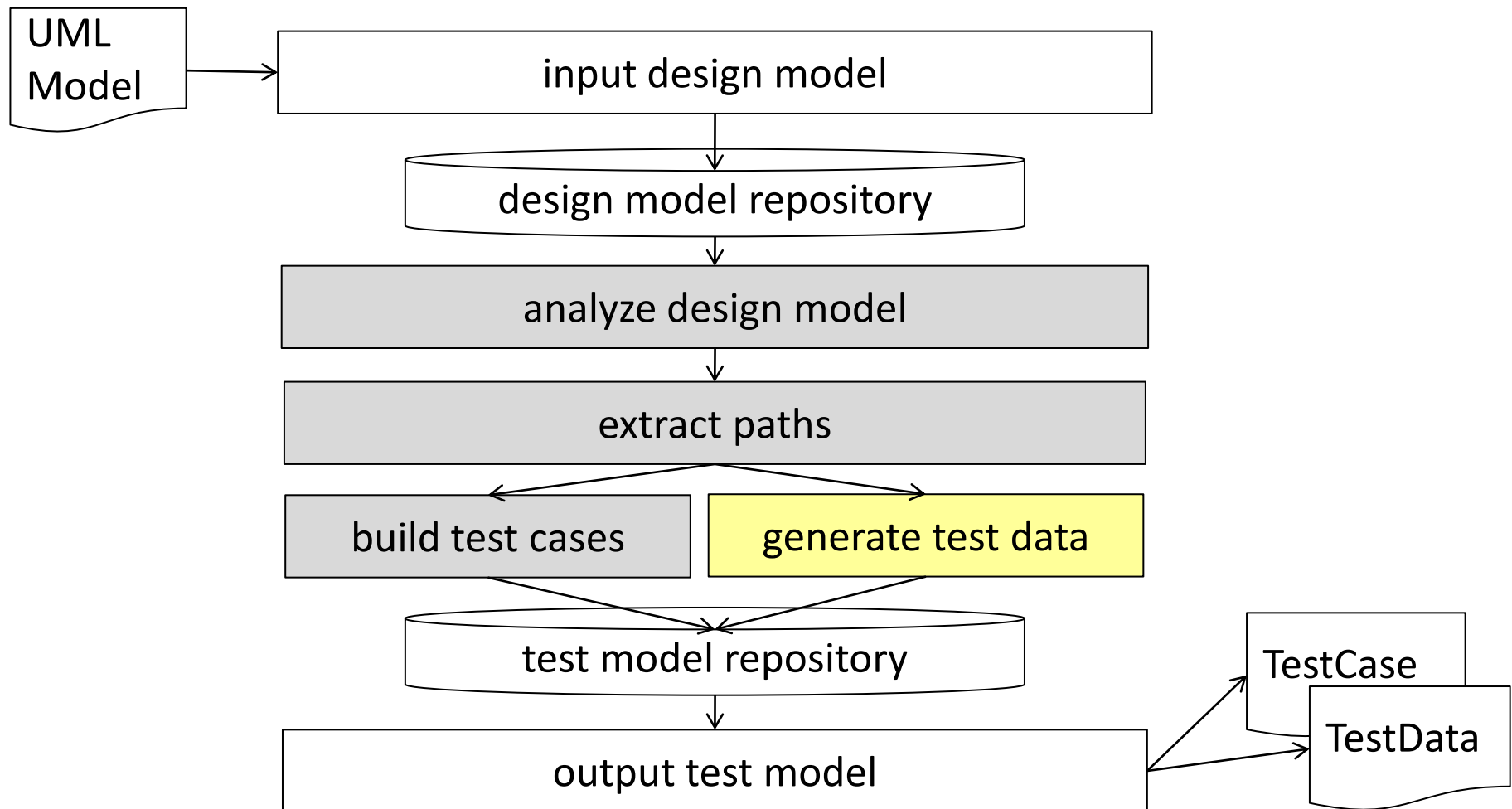


4. Organize test cases as a test model

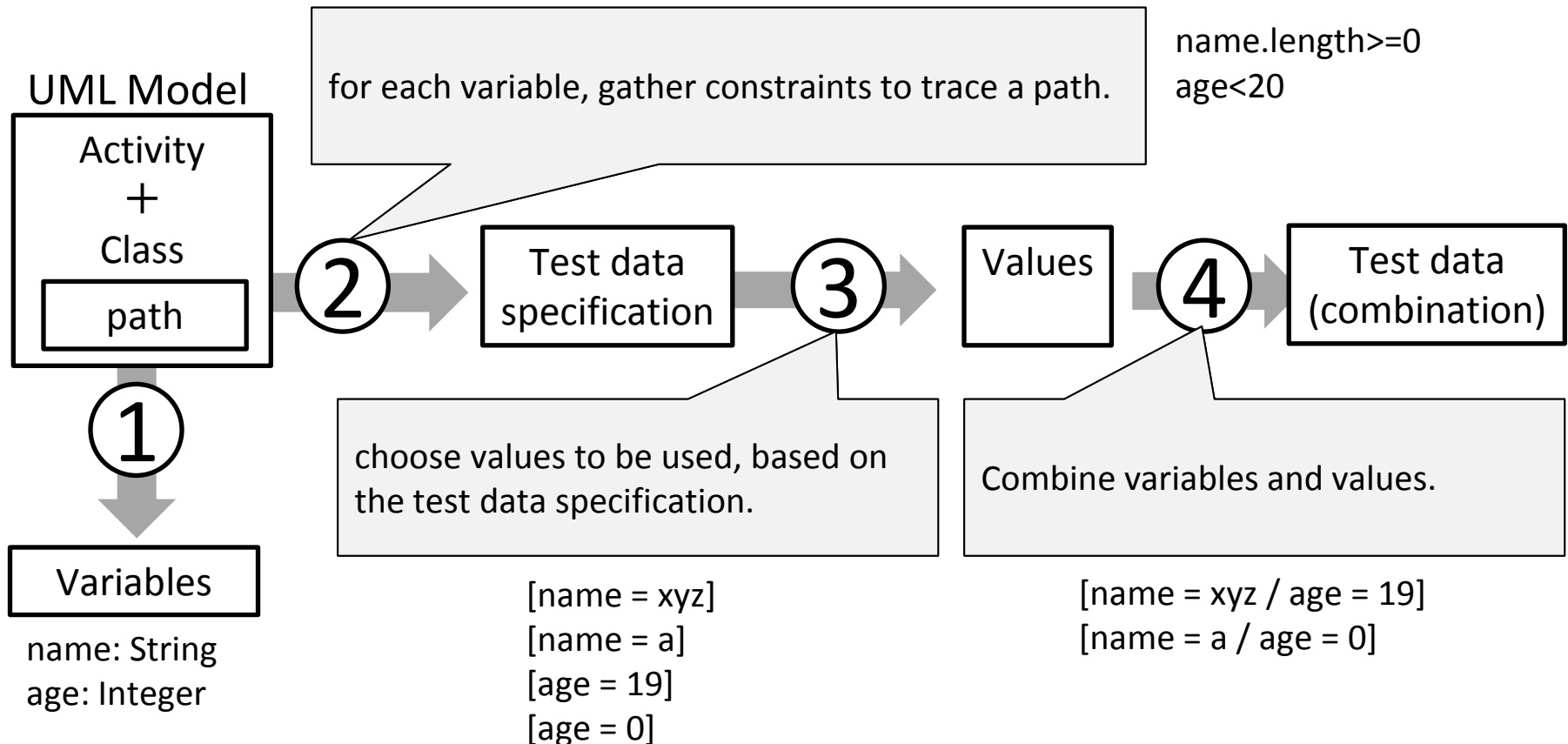
Can be categorized by test level or viewpoints



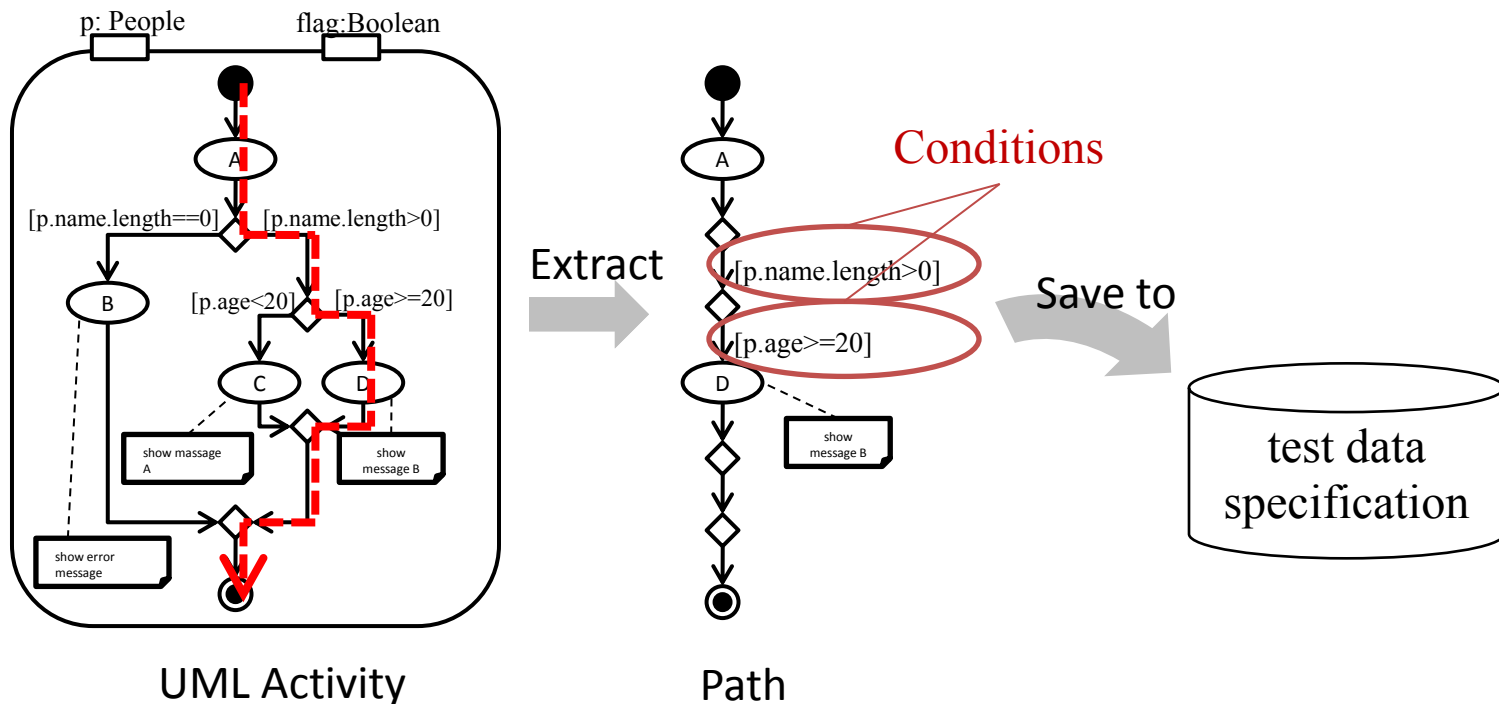
Test Data Generation



- In order to achieve high “input space coverage”, we thoroughly obtain the variables and the values which constitute the test data.
- The generation process has 4 steps.

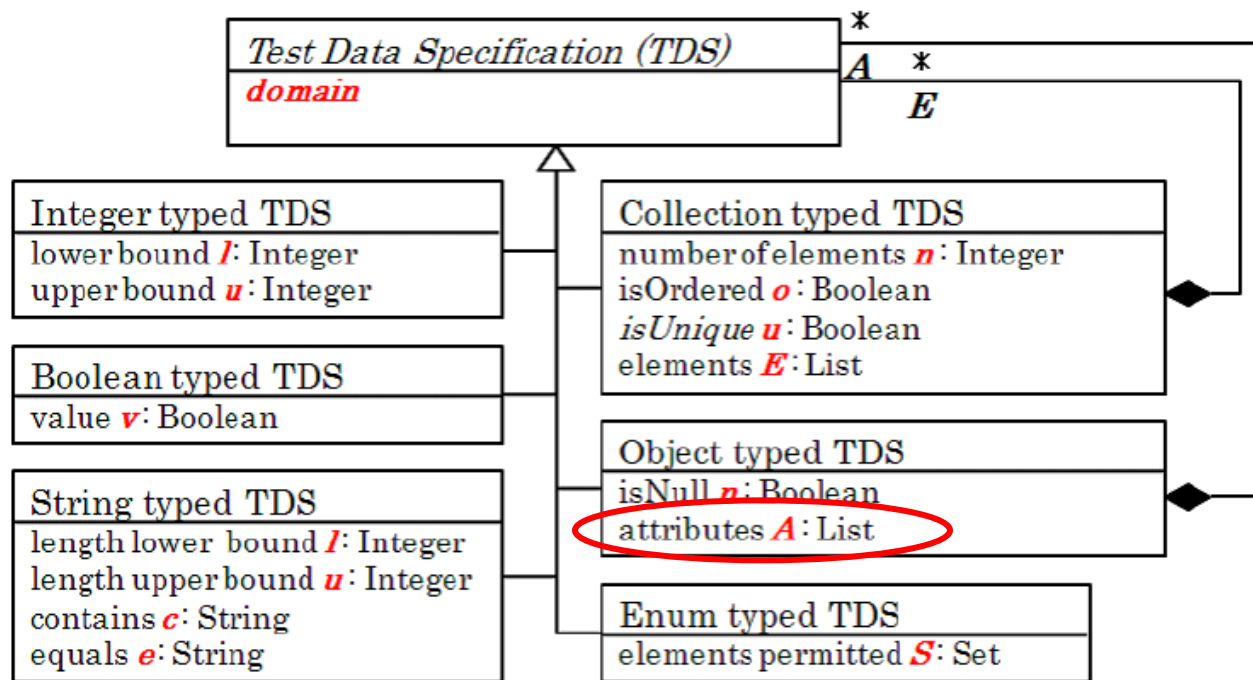


- ① Extraction of the variables
 - activity parameter node
- ② Creation of the “test data specification”
 - which save the constraints for the variables.



② Generation of the “test data specification”

- Generated per each variable of each execution path, based on the idea of the domain reduction
- Different “domain” for different data type.
- Domain of Object data type, contains specification of its own attributes: deal with hierarchy structure



③ Generation of the values

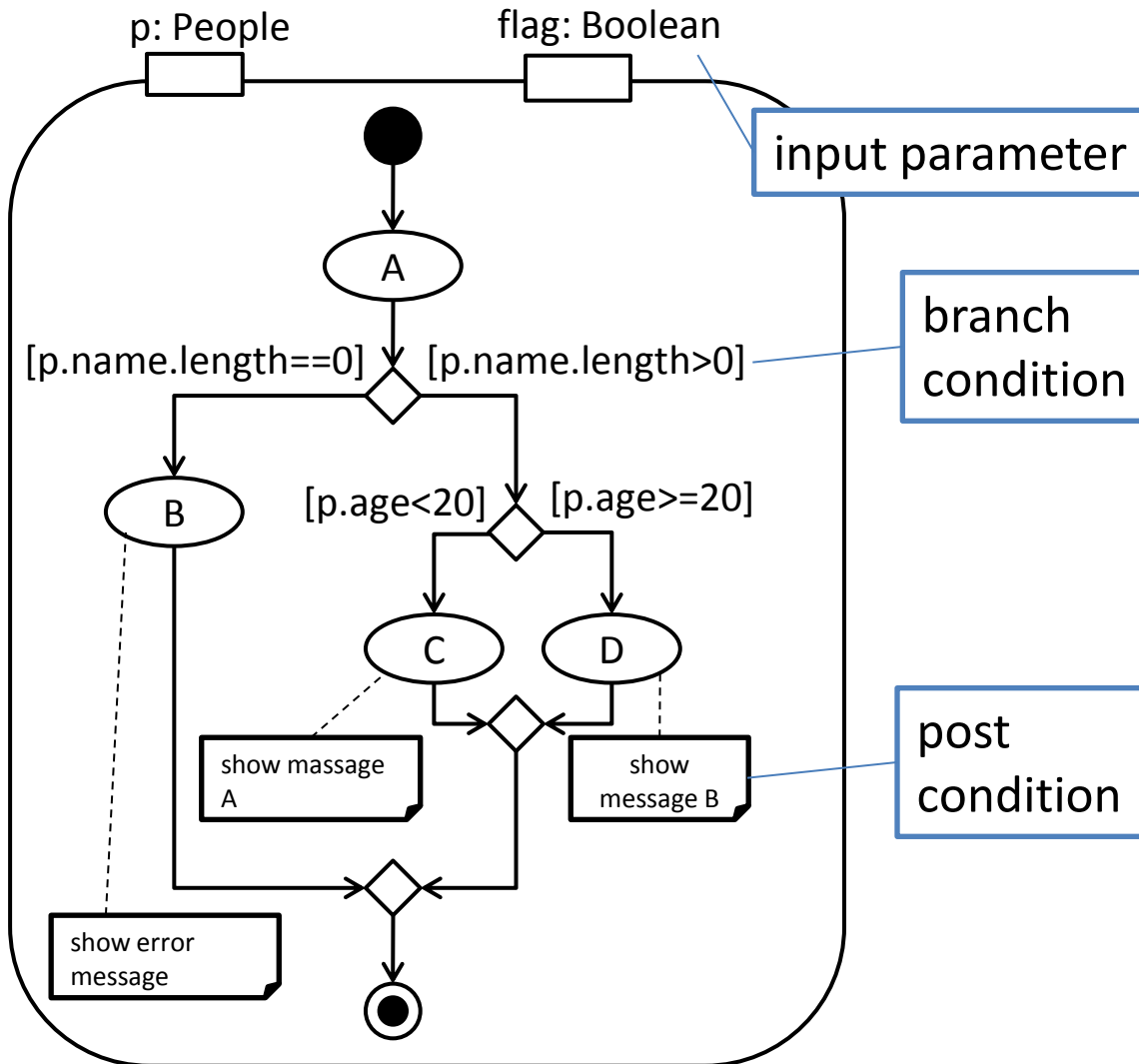
- boundary value analysis
- normal values / abnormal values
- as much as one “abnormal value” in a “leaf node” means the entire object is an abnormal value.

④ Combination of the values

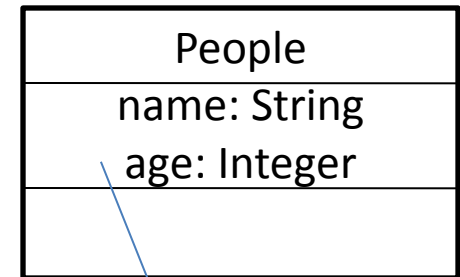
- Normal Inputs : minimum combination
- Abnormal Inputs : contains only one abnormal value

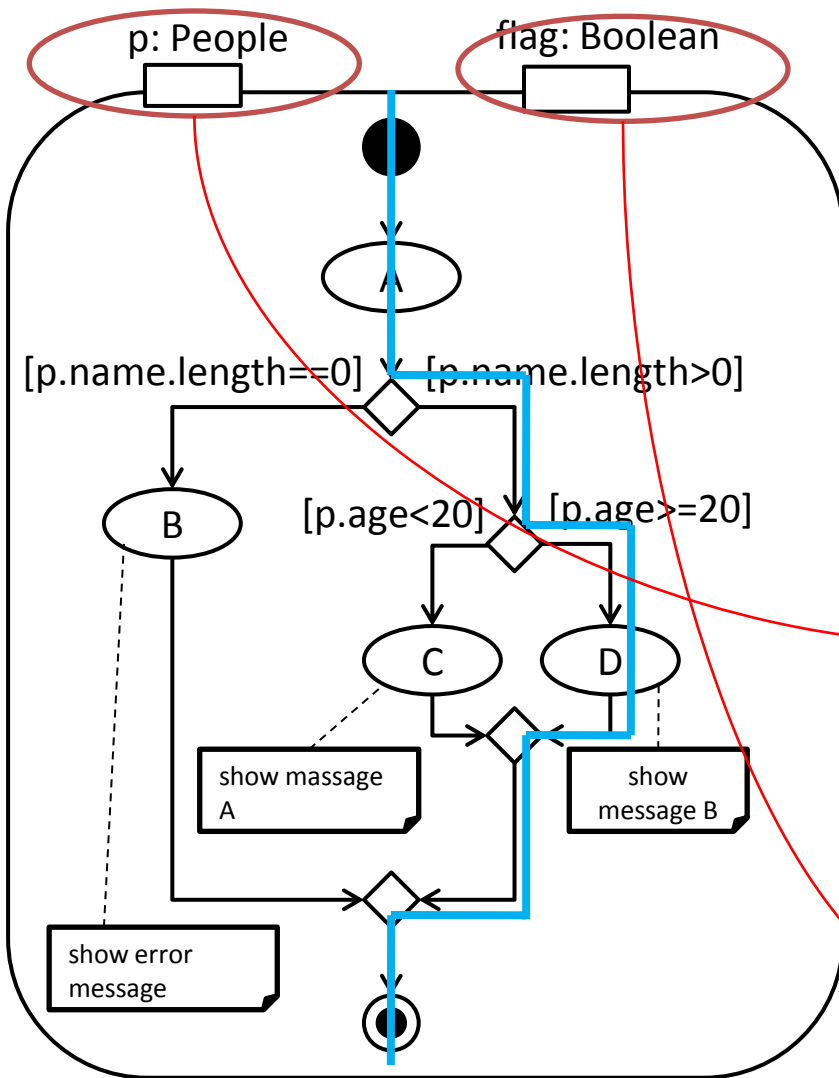
- Cannot handle variables except the input parameters
 - temporary variables, global variables
 - Need more data modeling.
- Cannot handle conditions describing:
 - the dependence between two or more variables.
 - Need dynamic domain reduction?
 - variables overwritten or changed.
 - Need symbolic execution or simulation?
- Inefficient combinations
 - just use all values generated is not good enough.
 - Need pair-wise methods?

UML Activity



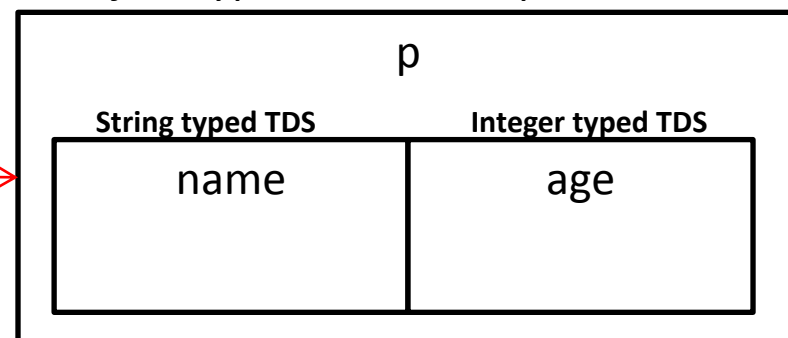
UML Class





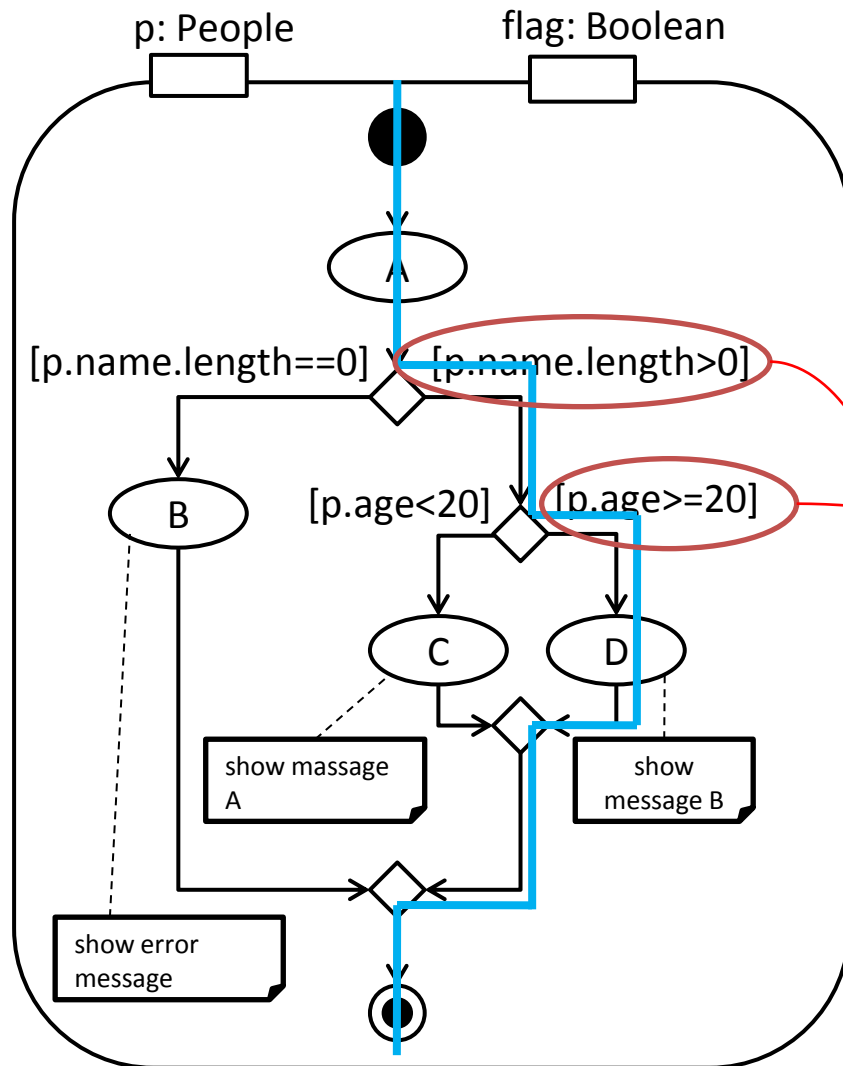
For each path,
Extract Input parameter as variables,
make and initialize a test data specification
depends on the variable's type

Object typed test data specification



Boolean typed test data specification





Update the test data specification, based on the branch condition

Object typed test data specification

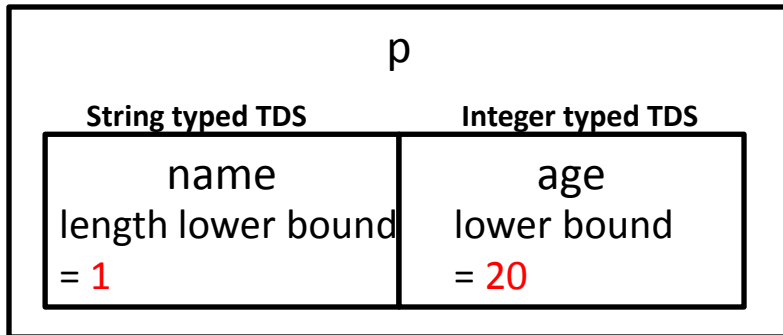
p	
String typed TDS	Integer typed TDS
name	age
length lower bound	lower bound
= 1	= 20

Boolean typed test data specification

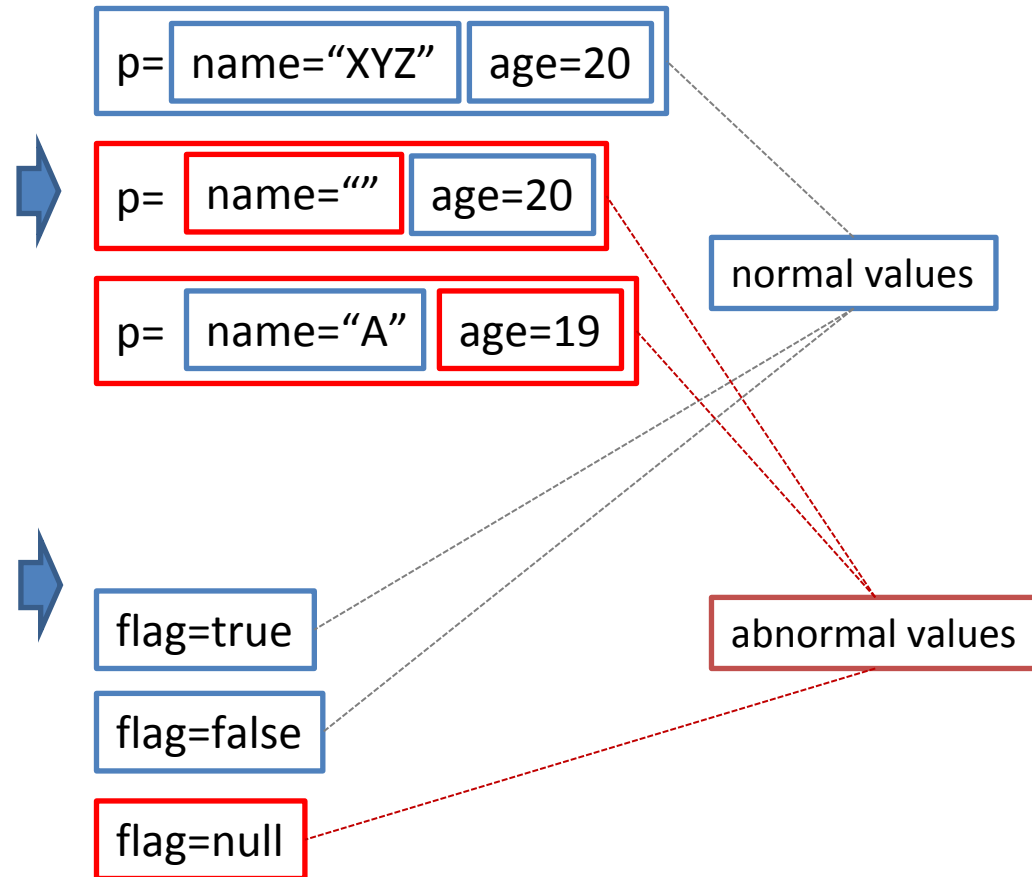
flag

Generate normal values and abnormal values from the test data specification.

Object typed test data specification

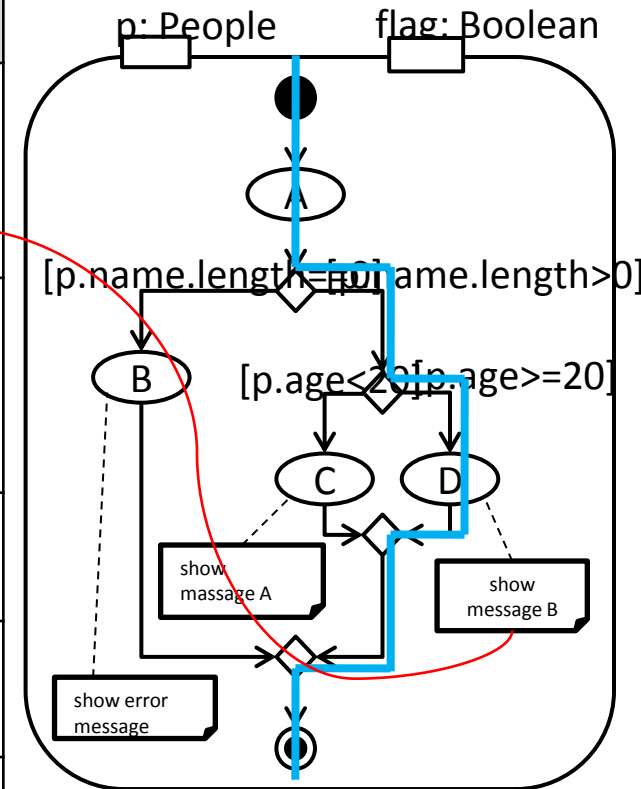


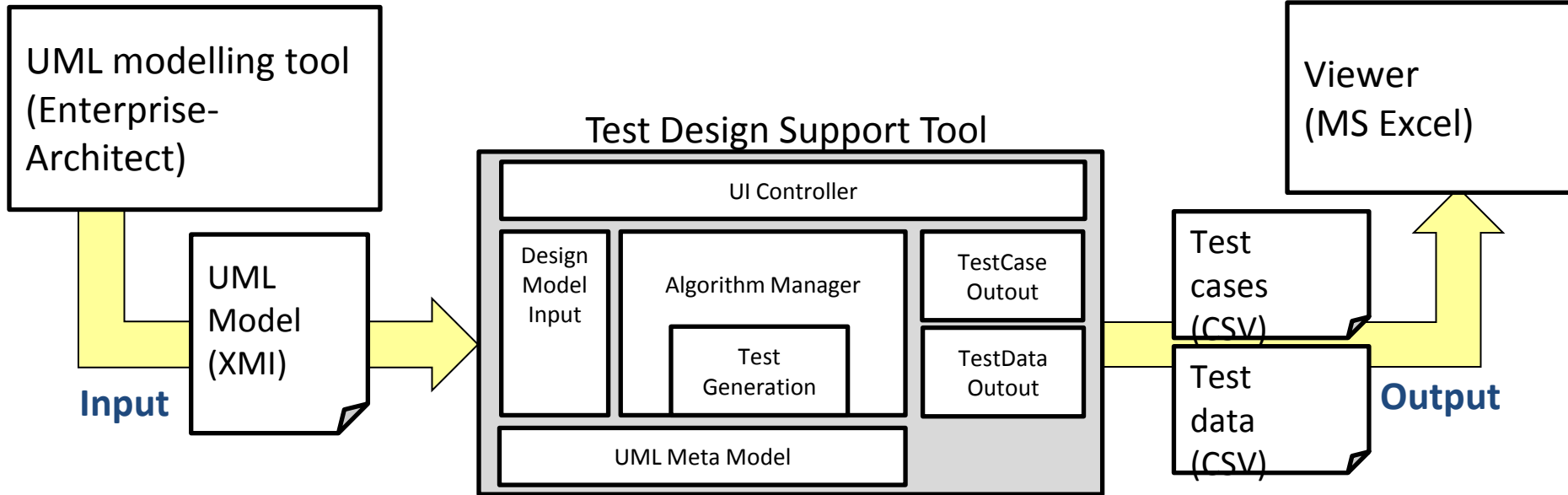
Boolean typed test data specification



An input value is a combination of the values set to variables respectively.
 An expected result is from user defined postcondition of the path.

ID	Input value	Expected result
1	p= name="XYZ" age=20 flag=true	show message B
2	p= name="XYZ" age=20 flag=false	show message B
3	p= name="" age=20 flag=true	indefinite
4	p= name="XYZ" age=20 flag=null	indefinite
	...	





Test cases

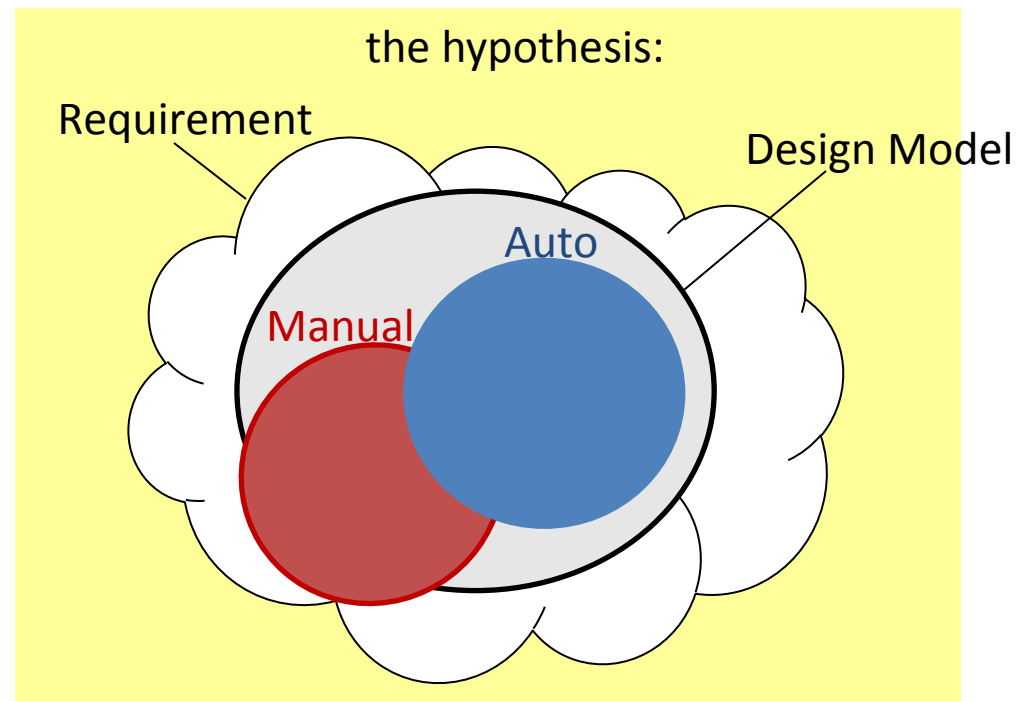
root	EA_Model	DPC	Components	LineItem	setQuantityShipped	PathNo0	test_setQuantityShipped_PathNo0_DataNo0
							test_setQuantityShipped_PathNo0_DataNo1
							test_setQuantityShipped_PathNo0_DataNo2
					setLocale	PathNo0	test_setLocale_PathNo0_DataNo0
							test_setLocale_PathNo0_DataNo1
							test_setLocale_PathNo0_DataNo2
							test_setLocale_PathNo0_DataNo3

Sample

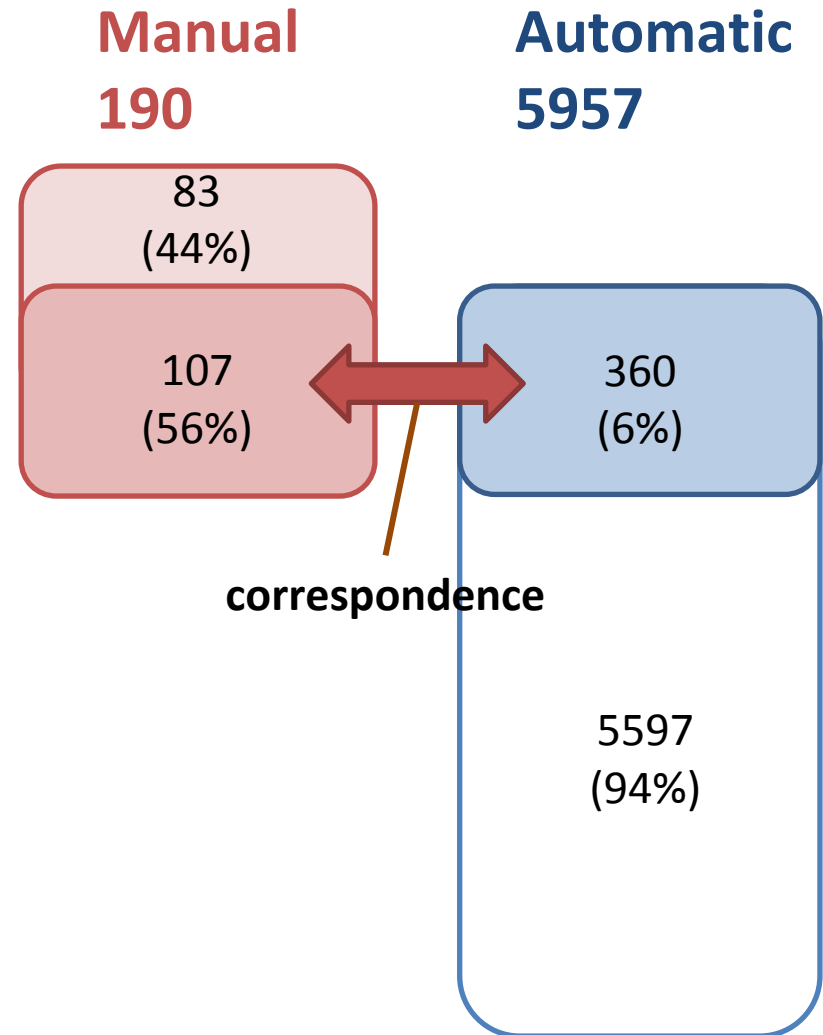
Test data

Path_Name	TestData_ID	status:OrderStatusNames	orderId:String	Expected_Status	Is_Normal	Test_Result
PathNo0	DataNo0	status:OrderStatusNames = APPROVED	orderId:String = 任意文字列(999文字)	changedOrderが追加されていること。	TRUE	OK / NG
PathNo0	DataNo1	status:OrderStatusNames = SHIPPED_PART	orderId:String = 任意文字列(1文字)	changedOrderが追加されていること。	TRUE	OK / NG
PathNo0	DataNo2	status:OrderStatusNames = DENIED	orderId:String = 任意文字列(1000文字)	changedOrderが追加されていること。	TRUE	OK / NG
PathNo0	DataNo3	status:OrderStatusNames = PENDING	orderId:String = 任意文字列(0文字)	INDEFINITE	FALSE	OK / NG
PathNo0	DataNo4	status:OrderStatusNames = COMPLETED	orderId:String = 任意文字列(999文字)	INDEFINITE	FALSE	OK / NG

- Case study
 - One component of a shopping store application
 - small Java web application (about 4.8KLOC)
 - Manually build an design model in UML2.0
 - Compare the artifacts made from the same model,
 - Manually derived by one average skilled developer
 - Automatically generated with our proposed method
- Viewpoints:
 - manpower cost
 - SUT coverage, test density



- Number of test cases
 - Good:
 - 56% of manual test cases can be generated by the proposed approach
 - Bad:
 - the technique could not correspond to 44% of the manual derived test cases.
 - the number of generated test data is too large to be used in test execution.



- Manpower cost
 - By automation,
about 52% manpower can be reduced

		Manual	Automatic
Man-power	on test design(minute)	2330	0
	on other overheads(minute)	0	1123 *
Unit man-power	per testcase (man-minute)	12	3
	per KLOC (man-hour)	8	4

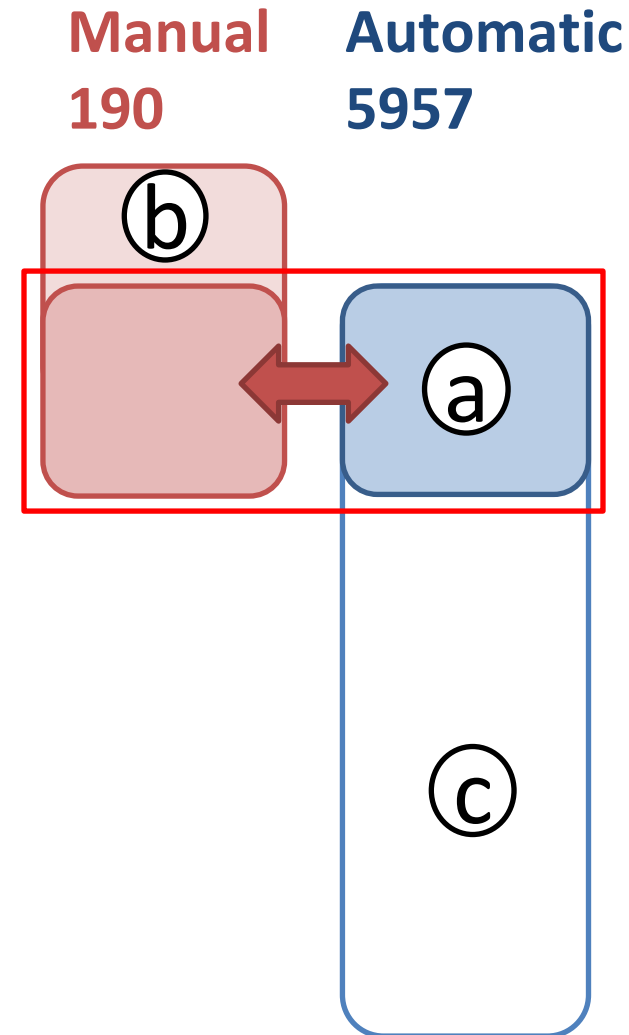
* The cost of retouching existing “rough sketch” UML model to a more detailed level, enough to generate test cases.

This assuming to be 50% of the overall cost of UML modeling.

- Test density and SUT coverage
 - By automation, test density increased to 1.9 times and the SUT coverage can be improved

Test density (testcase per KLOC)		40	75	
SUT Coverage	Structure		all paths	all paths
	Input space	values	normal inputs representative-values only	Normal/abnormal inputs boundary values/ representative values
		combinations	without clear rules	cover all values

- ① Test cases WITH correspondence
 - Most of automatic test cases with normal inputs
 - improvement in input space coverage
 - more variation of value
 - more exhaustive combination
- ② Manual test cases which could not be automatically generated
 - difficult to formalize semantics of the software
 - nested structure of activities
- ③ Automatic test cases which have no correspondence to manual test cases
 - most of test cases with abnormal inputs
 - The value of object data type has more variation than human usually consider



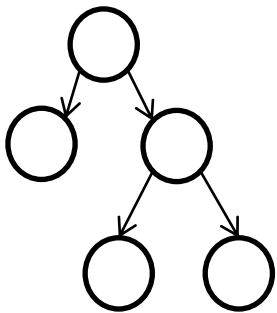
● What Next?

(b) How to deal with the knowledge not described in the UML model?

- decide a notation and describing them in the UML model.
- Or, extract from information formalized outside of the UML model.

(c) “many more” does not mean “better”.

- It is necessary to narrow down the number of test cases to a practicable level



- For example, combination of not all “leaf level” abnormal values, but the important ones only

- Small “toy application” ...
 - apply on larger, alive project
- Still, a lot of constraints!
 - other variables except the input parameters
 - conditions contains dependence between two or more factors.
 - value of the variable overwritten or changed along the execution path.
- Too many test cases generated?
 - cooperation with test execution automation tools such as xUnit.

- The proposed method extract test cases, and test data with hierarchical structure, from UML activity and class models.
- our goal was to...
 - improve “test design”, with low cost by automation
 - By introducing automatic generation, for A PART OF the test design work, higher SUT coverage and test density can be achieved with fewer man-power.
 - Though further improvement is required.
 - Need to apply other methods to solve the restrictions.
 - Need methods to handle the outside of UML model

- Our vision on model based testing
 - Totally automated (combine test design automation with test execution automation)
 - large number of test cases will not be problem.
- Gaps!
 - Still need human to check the detailed test result.
 - Test oracle?
 - Not all test cases can be generated automatically.
 - Cooperation with human.
 - Modeling is still difficult for normal developer.
 - Convert from “ordinary” design document?
 - Integrate into (or change?) the development process
 - Explain the merit of modeling and automated generation.

Merci!
Questions or comment?

zhang.xiaojing@lab.ntt.co.jp