PSSE 2007

Automatic Test Case Generation

Patricia Machado

DSC/UFCG

patricia@dsc.ufcg.edu.br

Augusto Sampaio

Cin/UFPE

acas@cin.ufpe.br

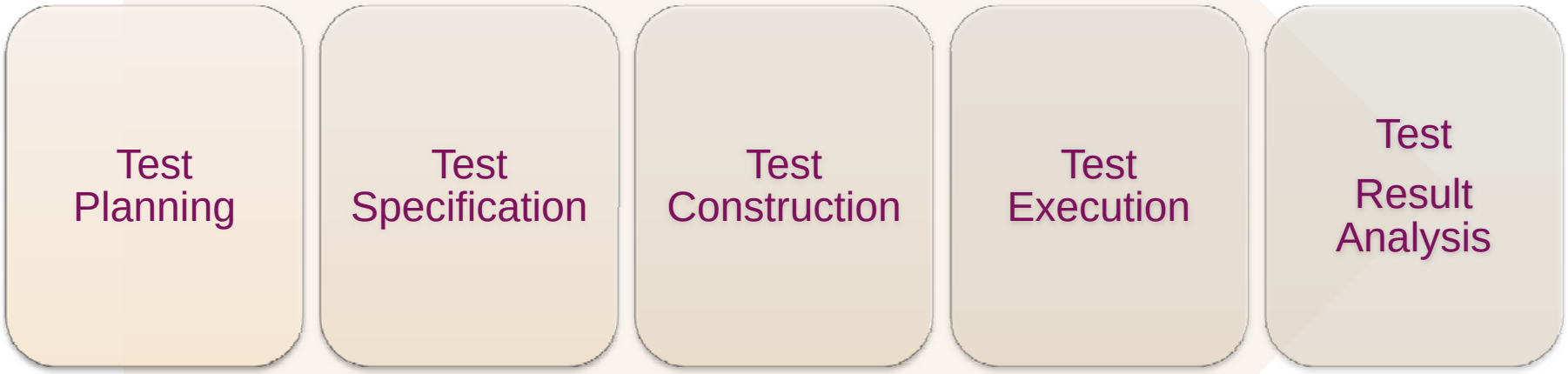
Recife / December 3rd-7th, 2007

Agenda

1. Introduction
2. Model-Based Testing
3. Test Models and Test Generation
4. Test Model Generation from Use Case Specifications
5. The TARGET Tool
6. Concluding Remarks and References

INTRODUCTION

Testing Processes



The Role of Abstraction

Test Planning	• What needs to be tested? • How much testing is needed?
Test Specification and Construction	• What are the scenarios to be exercised? • What are the behaviours to be observed? • What coverage is needed? • How to select relevant test cases?
Test Execution and Analysis	• How test execution is conducted? • How test results are observed? • What conclusions can be reached?

The Use of Models for Testing

- Models can formalise the use of abstraction:
  - High level models can represent general functionalities for supporting Test Planning
  - Test Cases can be automatically generated from behavioural models of an application:
    - Systematic coverage
    - Increased productivity and reliability
  - Result analysis can be more effective and accurate

Automatic Test Generation

- Test cases are derived from a specification model according to a given coverage criteria and testing goal;
- The specification model is an abstract representation of the behaviour we wish to test;
- Test cases are defined be exercised in a System Under Test (SUT) that is often constrained to be modelled as a particular kind of concrete model (*Test Hypothesis*);
- Test results are evaluated and analysed according to decision procedures often called *oracles* (automatic test evaluation);

Automatic Test Generation

- Test cases can be generated either in a test description language (TTCN, CNL) or in a target programming language (test execution frameworks);
- Test execution infra-structure can also be generated from the overall development models of an application;
- Test generation often follows pattern of test execution architecture and test documentation;
- Full code generation can be achieved (Model-driven engineering).

Automatic Test Case Generation in This Course

- In this course, we focus on:
  - Test generation from transition systems and process algebra;
  - Automatic test generation, selection and documentation, not including test code generation;
  - Model-based testing;
  - Test model generation from requirements documents;
  - Mobile phone applications domain.

MODEL BASED TESTING

- Test Models
- MBT in the Mobile Phone Applications Domain

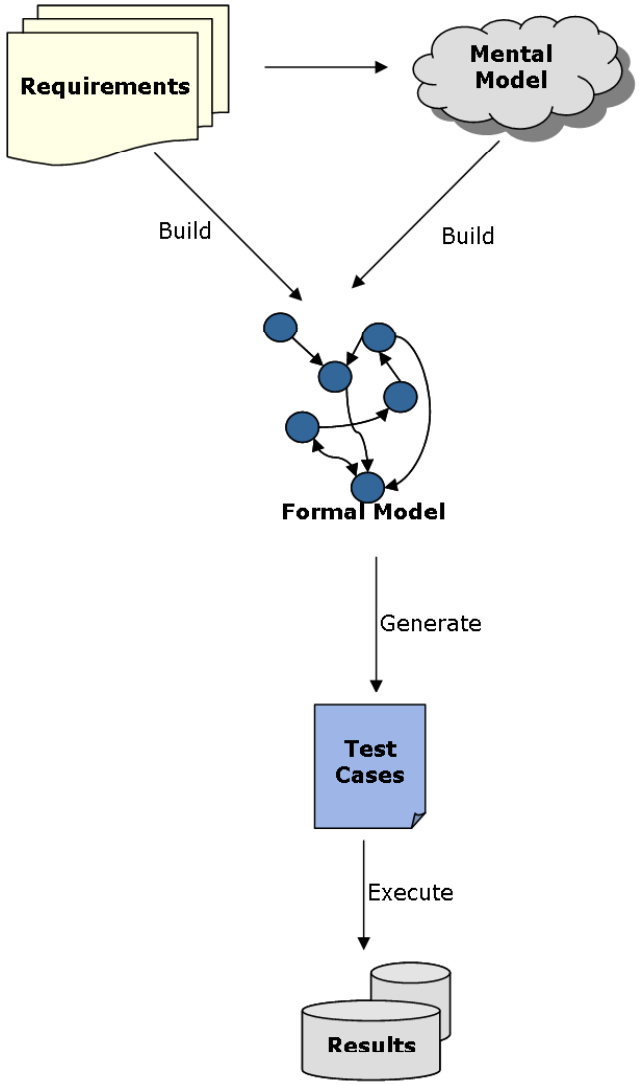
Model-Based Testing (MBT)

- MBT is a testing approach that makes use of models, usually named as test models, for representing the behaviour we want to test at a target implementation.
- From the test model, test cases are generated according to a given coverage goal.
- When formal models for generating test cases and abstracting details from the SUT are considered, this approach is commonly known as conformance testing.

Model-Based Testing (MBT)

- The main promises of MBT are:
  - Increased effectiveness in testing, by leading to focus and fair coverage of what needs to be tested;
  - Test costs reduction, by possible reuse of development artefacts and automation of testing artefacts generation;
  - Increased reliability in testing due to the fact that most of the test artefacts are automatically generated in contrast with manually coding them.

MBT Process Model



Centro de Informática U·F·P·E

MOTOROLA intelligence everywhere™
Mobile Devices Software Group

MBT and Test Case Generation

- Test case generation in an MBT process usually demands:
  - A good understanding of the SUT;
  - An appropriate model for representing the requirements for the system that suits the application domain and testing goals (can test hypothesis be met?);
  - A systematic procedure for building the model either by hand or automatic generation;
  - Test generation algorithms that can be automatically applied according to coverage and test selection criteria;
  - A validation strategy from which the test cases generated can be assessed regarding whether they are consistent and feasible w.r.t. the SUT.

MODEL BASED TESTING

- Test Models
- MBT in the Mobile Phone Applications Domain

Test Models

- Abstract representations that are developed for test case generation;
- They can be built from scratch or derived from development models;
- Test models must support both manual and automatic test selection;
- Test models must be testable;
- Test models should be based on a well-found test theory.

Test Model Requirements

- Be a complete and accurate representation of all features to be tested, allowing valid coverage conclusions;
- Abstract detail that is not the focus of the test and would make the cost of testing prohibitive;
- Preserve detail that is key for revealing faults of interest and also for setting up test execution and make test oracle generation possible;
- Represent all stimulus and states that are visible externally.

Models for Test Case Generation

Functional Testing

- State Machines;
- Markov Chains;
- UML models;
- Formal models in general.

Structural Testing

- Control graphs and data flow graphs
- Dependency graphs

What is a Test Case?

- A test case represents a particular situation we wish to exercise in a SUT that is defined to suit a particular objective such as:
  - To exercise a particular program path;
  - To check compliance with a requirement.
- Test cases are often represented by:
  - A set of input values or stimuli (test points)
  - A set of execution pre-conditions
  - A set of expected results or observations
  - A set ot execution postconditions

How Test Cases are generated from Models?

- By choosing paths or sequences of events or symbols according to coverage criteria metrics (all nodes, all arcs, all paths, and so on);
- By solving constraints, when test cases are to be derived from logic properties;
- Test cases must be finite and are often limited to a given length;
- Test cases must be determinist and feasible;
- For test case execution, test data selection may be required.

MODEL BASED TESTING

- Test Models
- MBT in the Mobile Phone Applications Domain

Feature

- A feature is a clustering of individual requirements that describe a cohesive, identifiable unit of functionality;
- Mobile phone applications are composed of a number of features that are usually highly interactive, having their flow of execution guided mostly by external input;
- Mobile phone applications have two specific types of testing:
  - **Feature testing** is the process of validating feature requirements by testing;
  - **Feature interaction testing** is the process of validating interaction of features that compose an application.

Feature Testing

- Features need to be thoroughly tested
- Manual testing
- Redundancy
- Domain requirements
- Time-to-market constraints
- Regression testing
- MBT is a promising approach

Feature Interaction Testing

- Interactions are defined here as:
  - scenarios where a feature functionality depends on another feature (*static interaction*);
  - Scenarios where there are combinations of independent behaviours (*dynamic interaction* or *interruption*).
- Examples:
  - Static Interaction - When we finish writing a message we can choose a contact from the phonebook to send the message (static interaction between the message feature and the phonebook feature)
  - Dynamic Interaction - When a user is composing a text message and suddenly a call arrives, characterising the interaction between the sending message and the incoming call features.

Feature Interaction Testing

- An effective MBT solution demands:
  - A test model where interruptions can be cost-effectively represented;
  - The test model must make feature interruption composition possible at different points of a flow of execution;
  - Selection strategies that can cope of the huge number of possible combinations of features and interruptions.

TEST MODELS AND TEST GENERATION

- Input-Output Labelled Transition Systems
- Annotated Labelled Transition Systems
- Process Algebra
- Markov Chains

TEST MODELS AND TEST GENERATION

- Input-Output Labelled Transition Systems
- Annotated Labelled Transition Systems
- Process Algebra
- Markov Chains

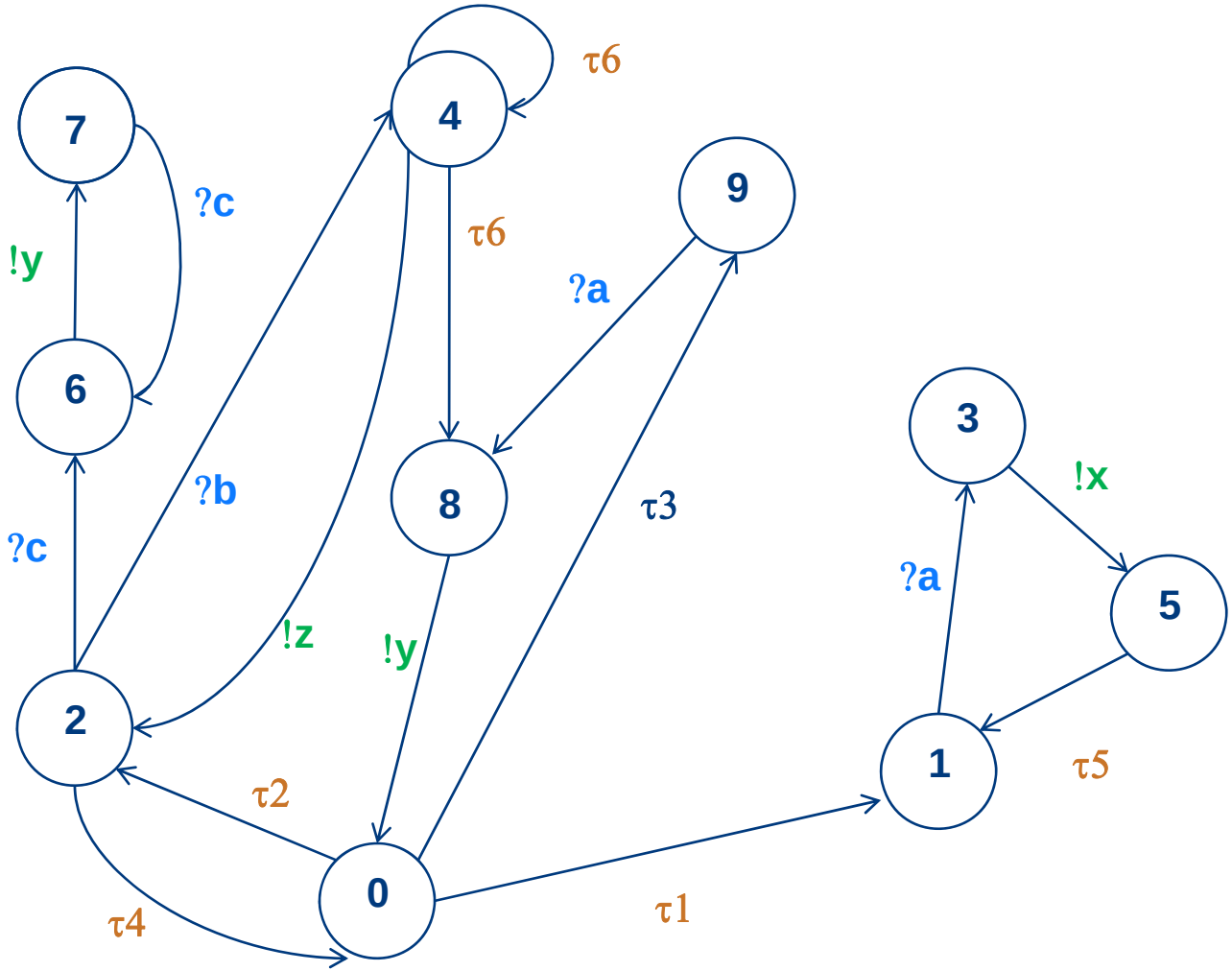
Input-Output Labelled Transition System (IOLTS)

- An IOLTS is a quadruple $M = (Q^M, A^M, \rightarrow_M, q_0^M)$, where:

  - Q^M is a finite non-empty set of states;
  - $q_0^M \in Q^M$ is the initial state;
  - $A^M = A_I^M \cup A_O^M \cup I^M$ is the alphabet of actions with A_I^M the input alphabet, A_O^M the output alphabet and I^M the internal actions alphabet;
  - $\rightarrow_M \subseteq Q^M \times A^M \times Q^M$ is the transition relation.

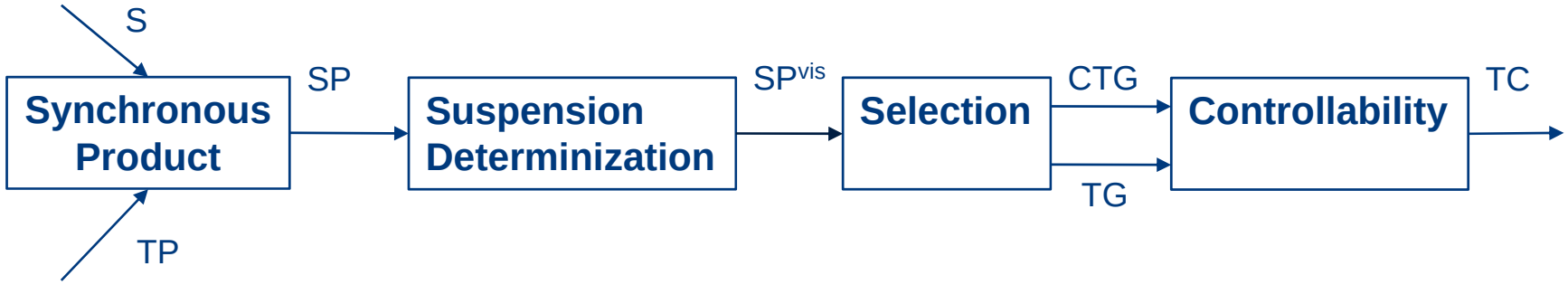
- This is the model adopted by the TGV tool

IOLTS



Test Generation with Verification technology (TGV)

- Automatic synthesis of conformance test cases for non-deterministic reactive systems;
- Based on verification techniques such as synchronous product, on-the-fly verification and traversal algorithms;
- Based on the ioco theory (Tretmans)



TGV Test Hypothesis

- Let S be a test model specification $\boldsymbol{S = (Q^S, A^S, \rightarrow_S, q_0^S)}$
- (Test Hypothesis) We assume that *SUT* can be modelled by an IOLTS $\boldsymbol{SUT = (Q^{SUT}, A^{SUT}, \rightarrow_{SUT}, q_0^{SUT})}$, that is:
  - Compatible with *S*: $A_I^S \subseteq A_I^{SUT}$ and $A_O^S \subseteq A_O^{SUT}$
  - *SUT* is input complete: all inputs are accepted, possibly after internal actions.

Conformance Relation

- Informally, *SUT* is **ioco**-correct w.r.t. *S* if:
  - *SUT* can never produce an output which could not have been produced by *S* in the same situation;
  - *SUT* may only be quiescent if *S* can do so.

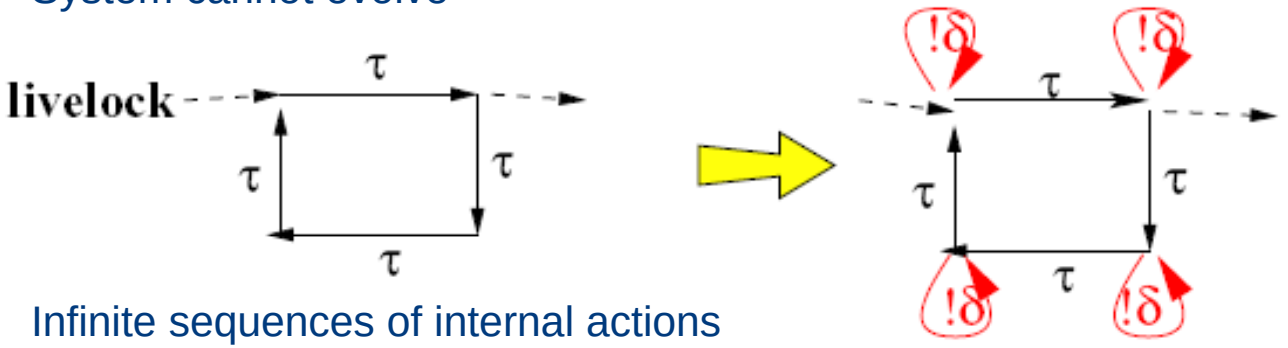
Δ

Quiescence

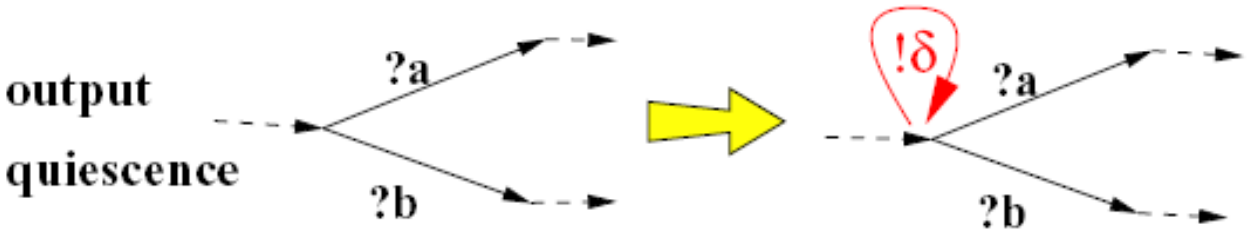
- The tester needs to obtain control over occurrences of input and output action in the SUT in order to conduct test execution;
- This requires the SUTs to be able to accept all input actions in any state (input-enabled or input complete);
- In this case, traditional deadlock states cannot exist;
- A weaker notion is needed:
  - states that cannot produce (further) output actions without the supply of an input actions are called *quiescent states*
  - Quiescence is treated as an observable event

Quiescence

System cannot evolve



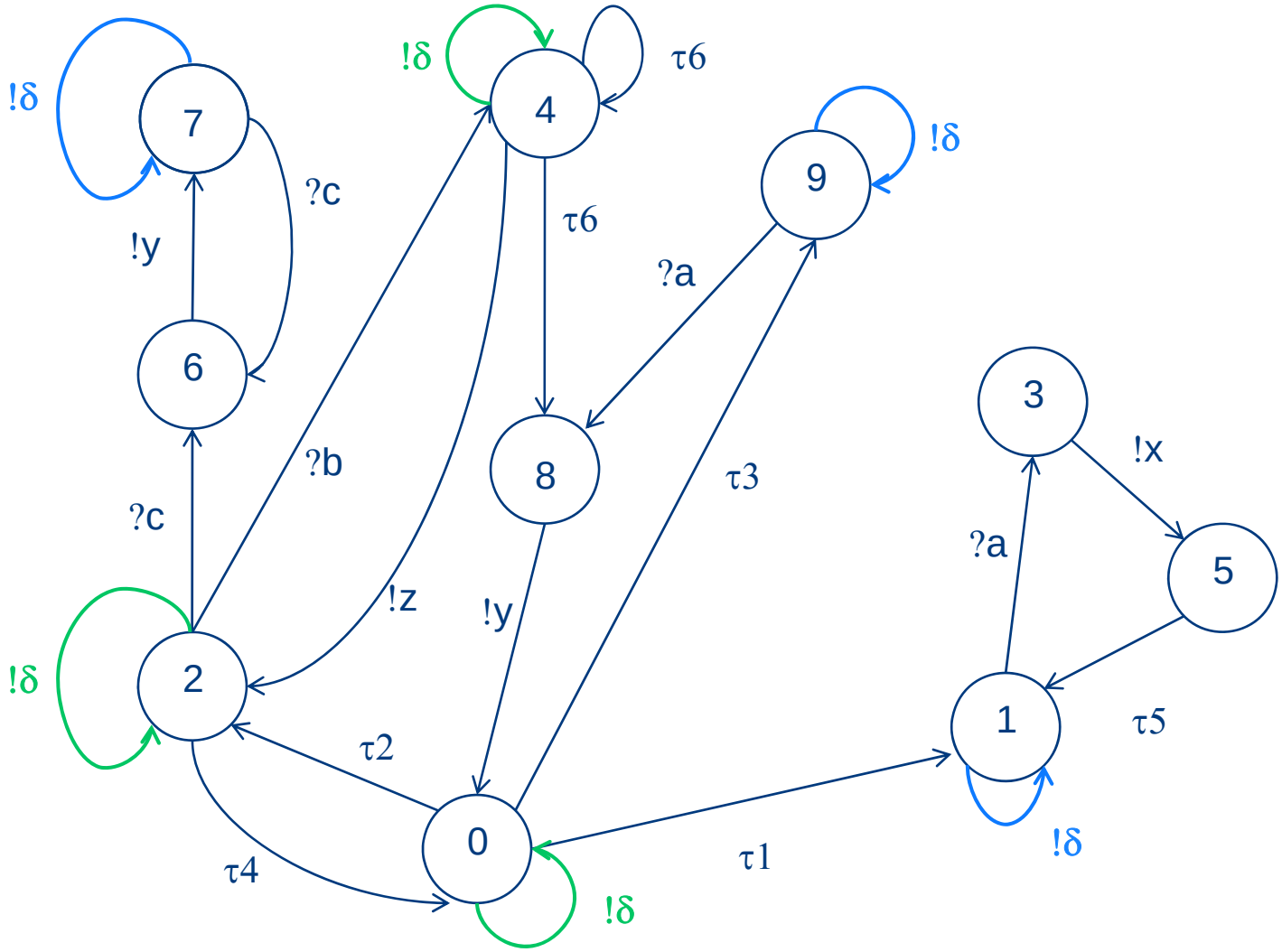
Infinite sequences of internal actions



Waiting only for an input from the enviroment

Figure from [JJ05]

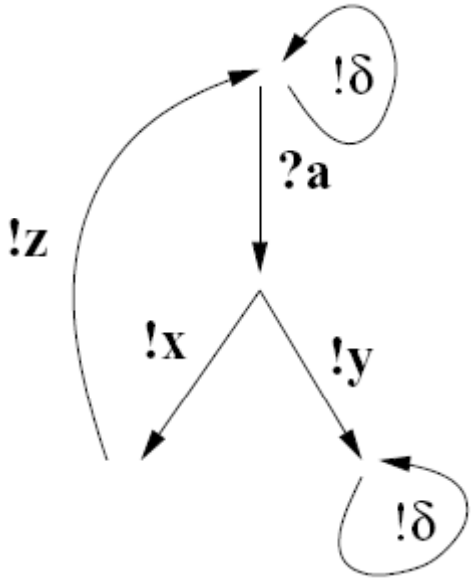
Suspension Automaton



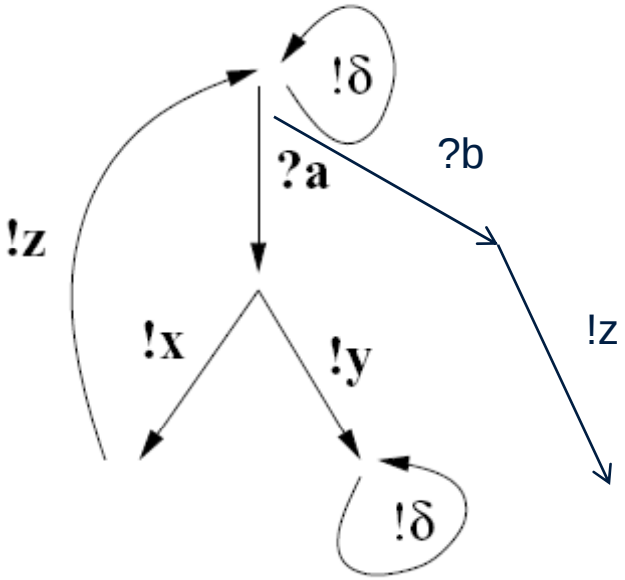
TGV Conformance Relation

- Conformance Relation:
  - *SUT* conforms to *S* if after each suspension trace of *S*, *SUT* exhibits only outputs and quiescences that are possible in *S*.

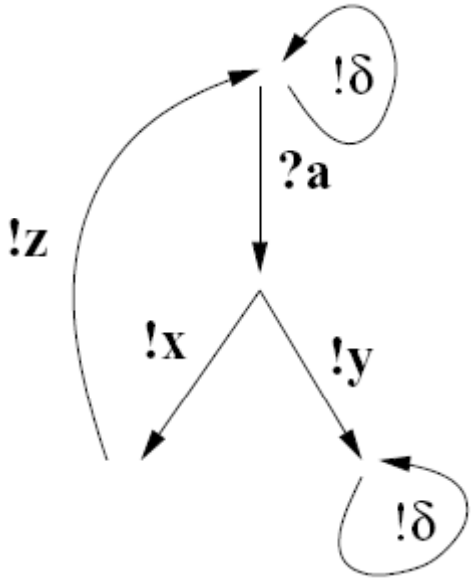
Conformance Relation



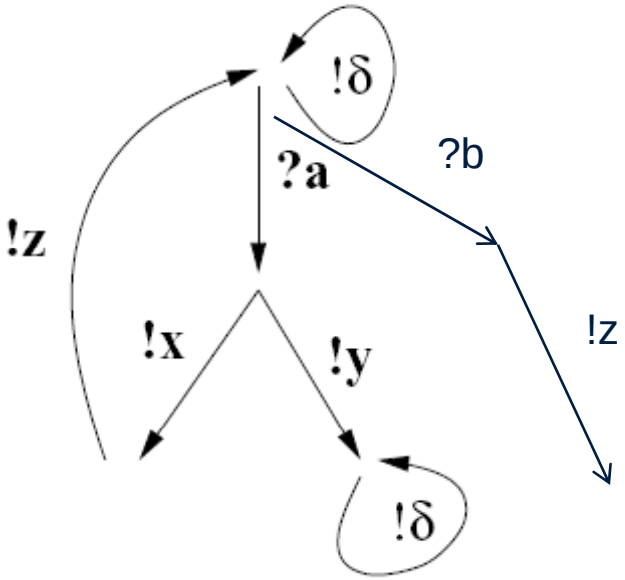
conforms ?



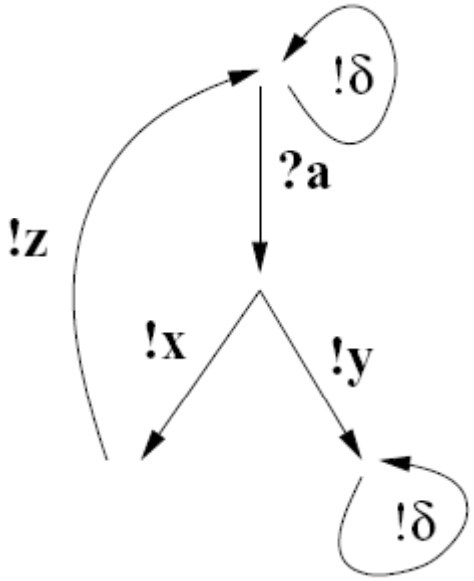
Conformance Relation



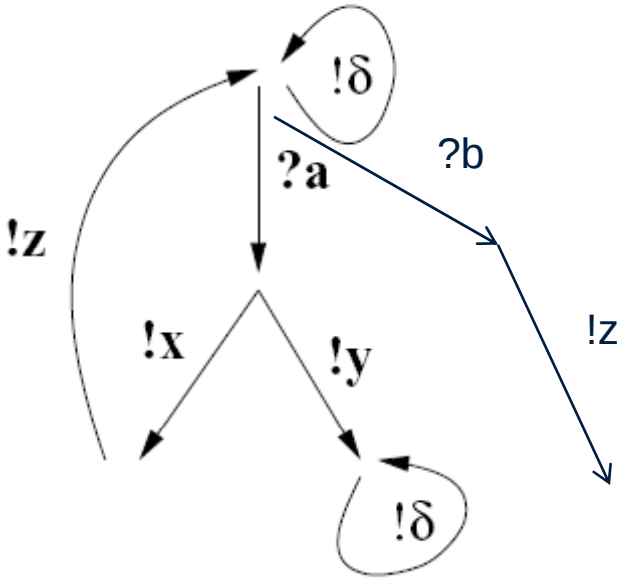
conforms



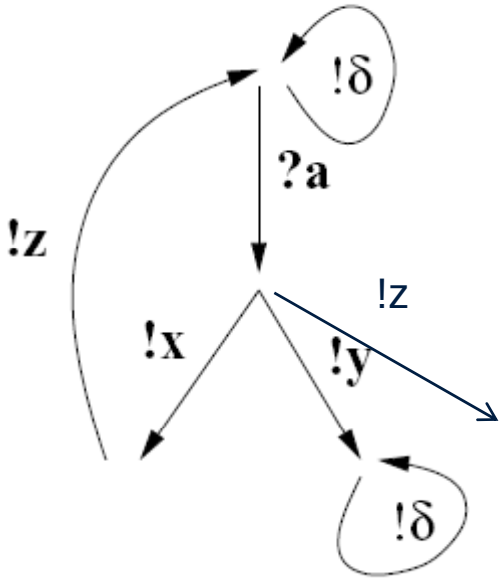
Conformance Relation



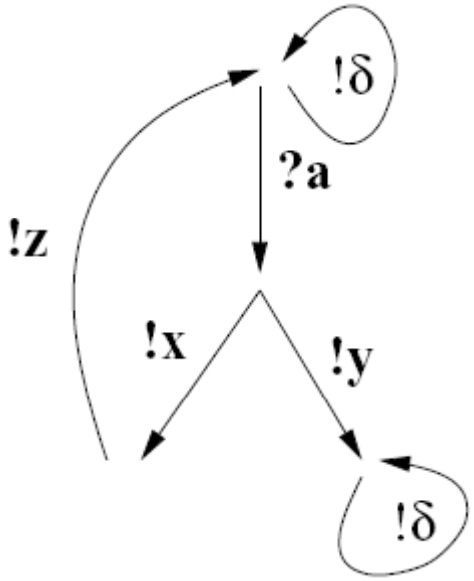
conforms



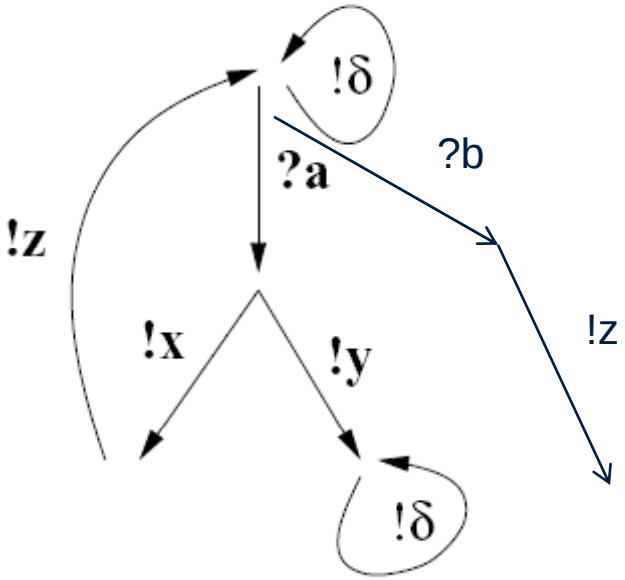
conforms ?



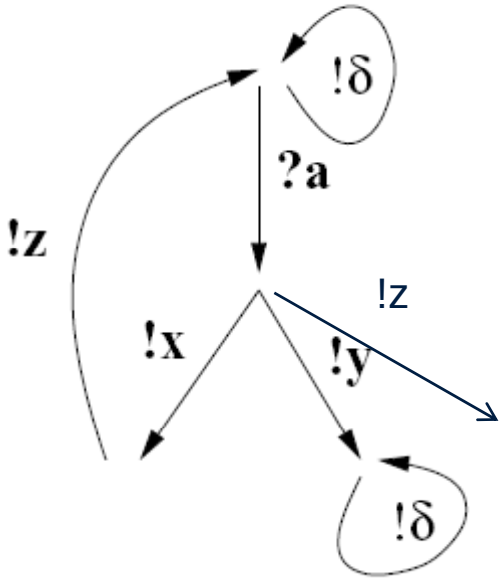
Conformance Relation



conforms



No



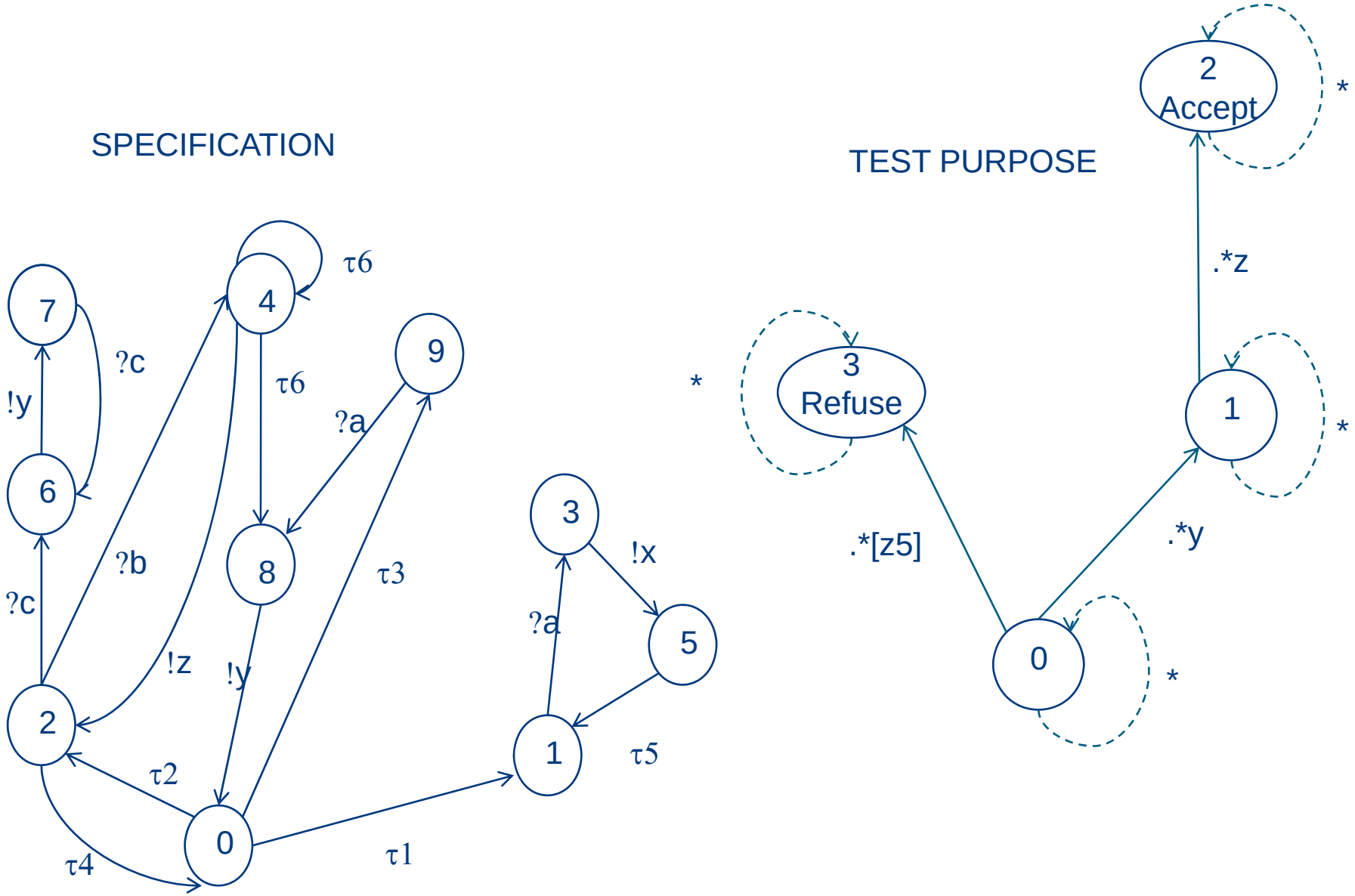
Test Purposes

- Target test generation at particular behaviours of the *S*.
- Testing for conformance is different from testing from test purposes (exhibition):
  - Conformance aims to accept/reject a given implementation;
  - Exhibition aims to observe a given behaviour;
  - If a desired behaviour is observed, than confidence on conformance may increase.
- Test suites for exhibition may be:
  - *e-complete* – can distinguish among all exhibiting and non-exhibiting *SUTs*.
  - *e-exhaustive* – can only detect non-exhibiting SUTs
  - *e-sound* – can only detect exhibiting SUTs

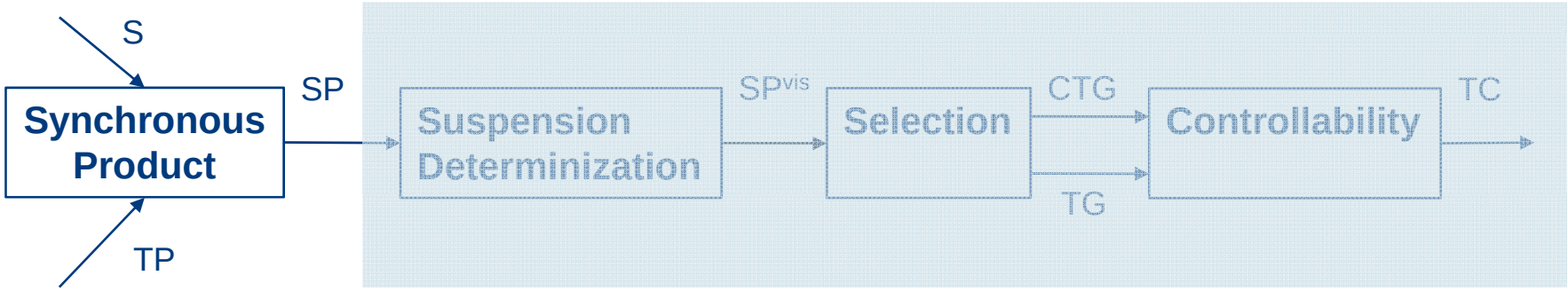
Test Purposes in TGV

- Modelled by IOLTSs that are deterministic and complete (each state allows all actions);
- The IOLTS have two marked states:
  - $Accept^{TP}$ – define sequences to be included in the test cases;
  - $Refuse^{TP}$ – define sequences not be included in the test case.

Test Purpose



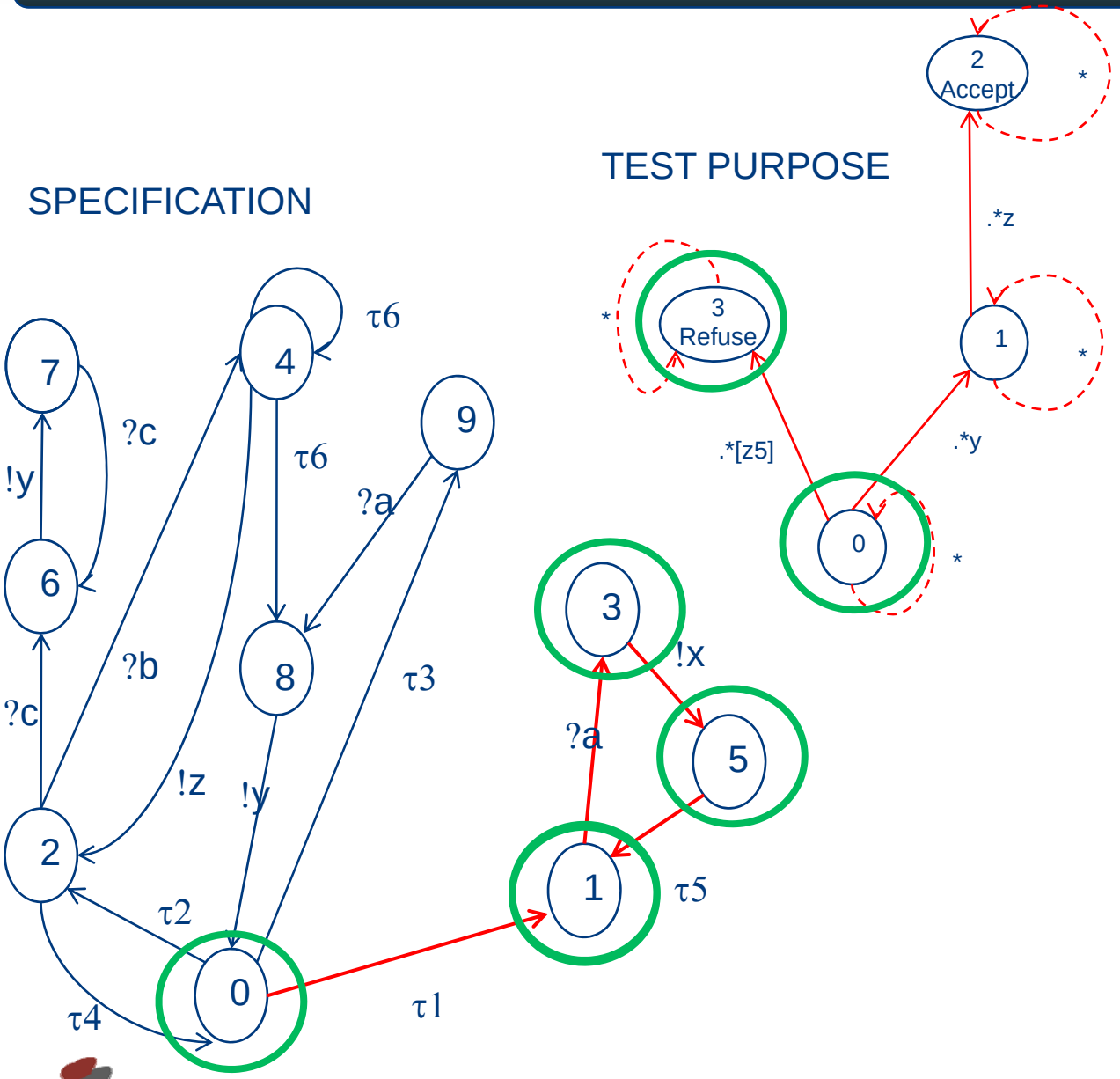
Synchronous Product

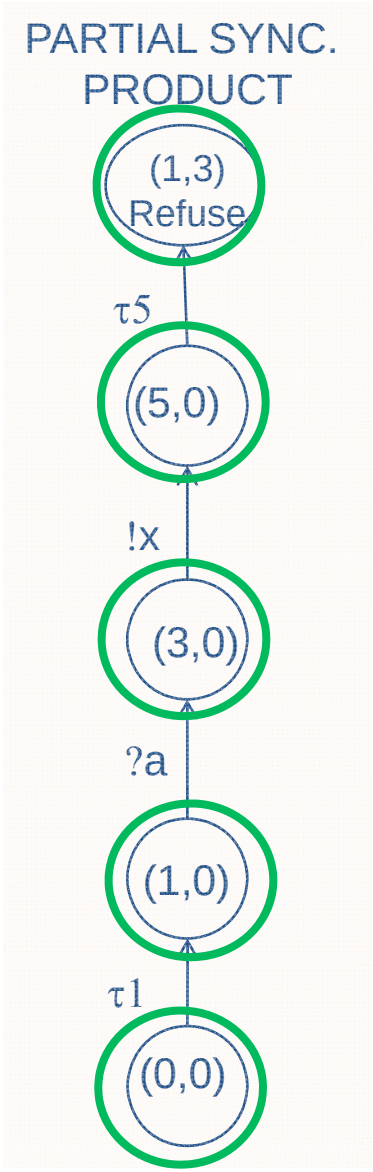


- Identify the behaviours of *S* accepted or refused by *TP*
- Intersection of *S* and *TP* that contains only states that are reachable from the initial states of *S* and *TP*
- All behaviours of *S* are preserved in *SP*
- (q_0^S,q_0^{TP}) is the initial state;
- The transition relation is defined as:

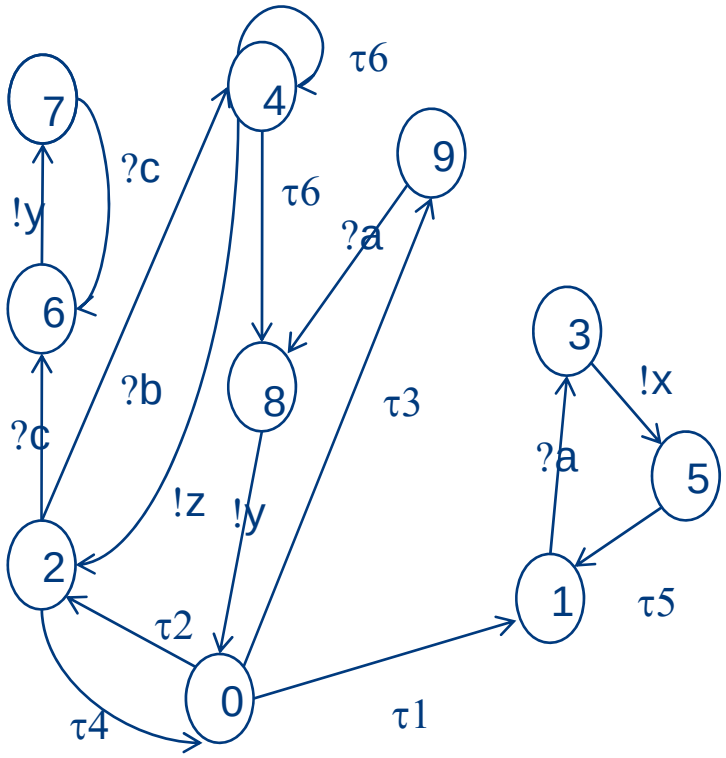
$$(q^S, q^{TP}) \xrightarrow{a}_{SP} (q'^S, q'^{TP}) \iff q^S \xrightarrow{a}_S q'^s \wedge q^{TP} \xrightarrow{a}_{TP} q'^{TP}$$

Synchronous Product





Synchronous Product



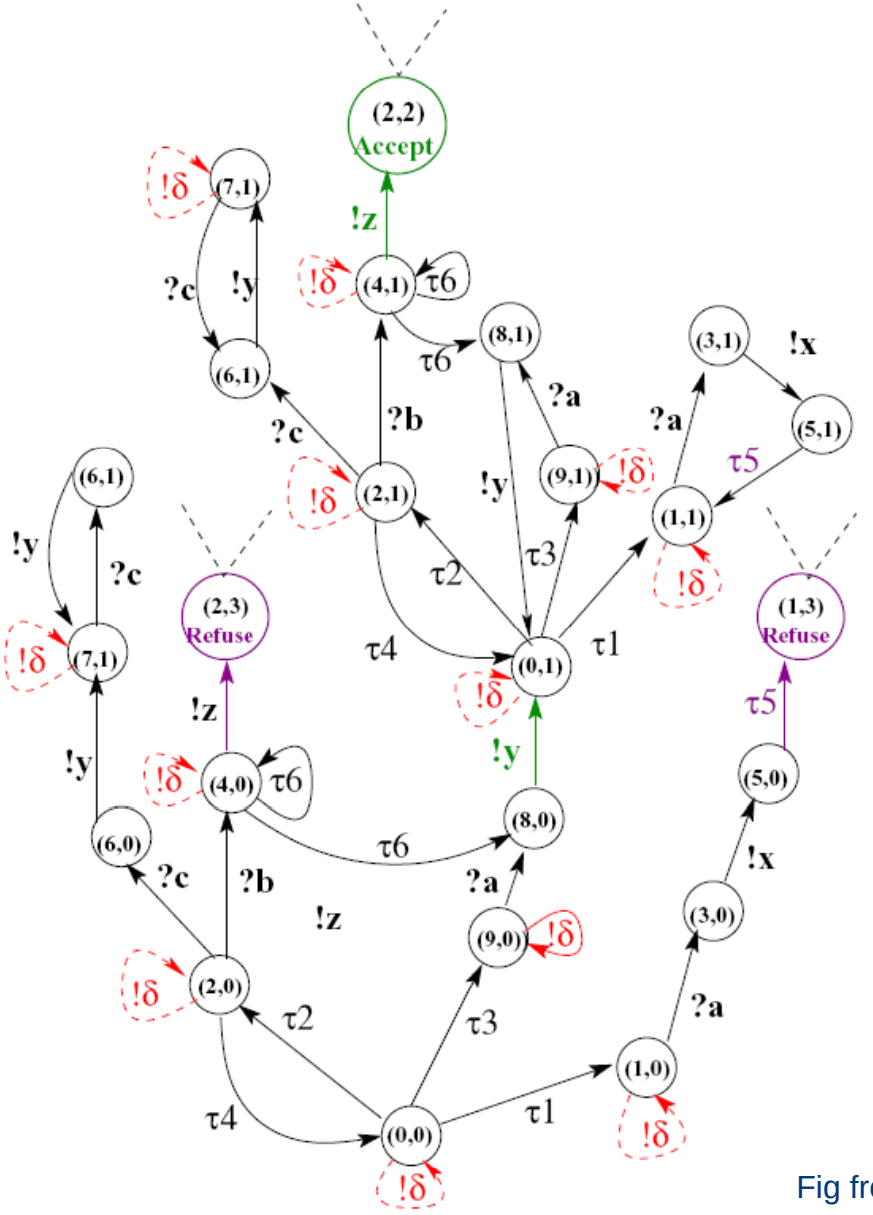
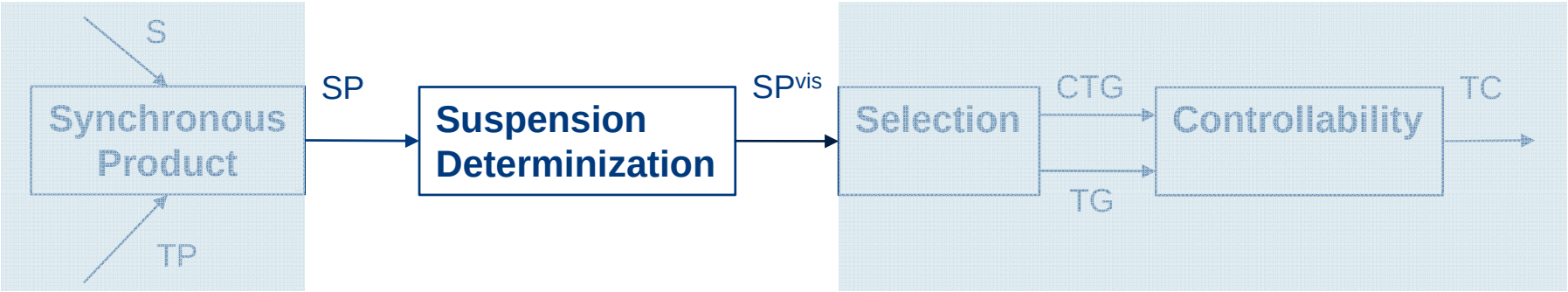


Fig from {JJ05}

Suspension Determinization



- SP^{VIS} = **det**(Δ(SP))
- **det** returns a derministic automaton that:
  - does not include internal actions;
  - states are meta-states defined for each reachable trace between meta-states.

Suspension Determinization

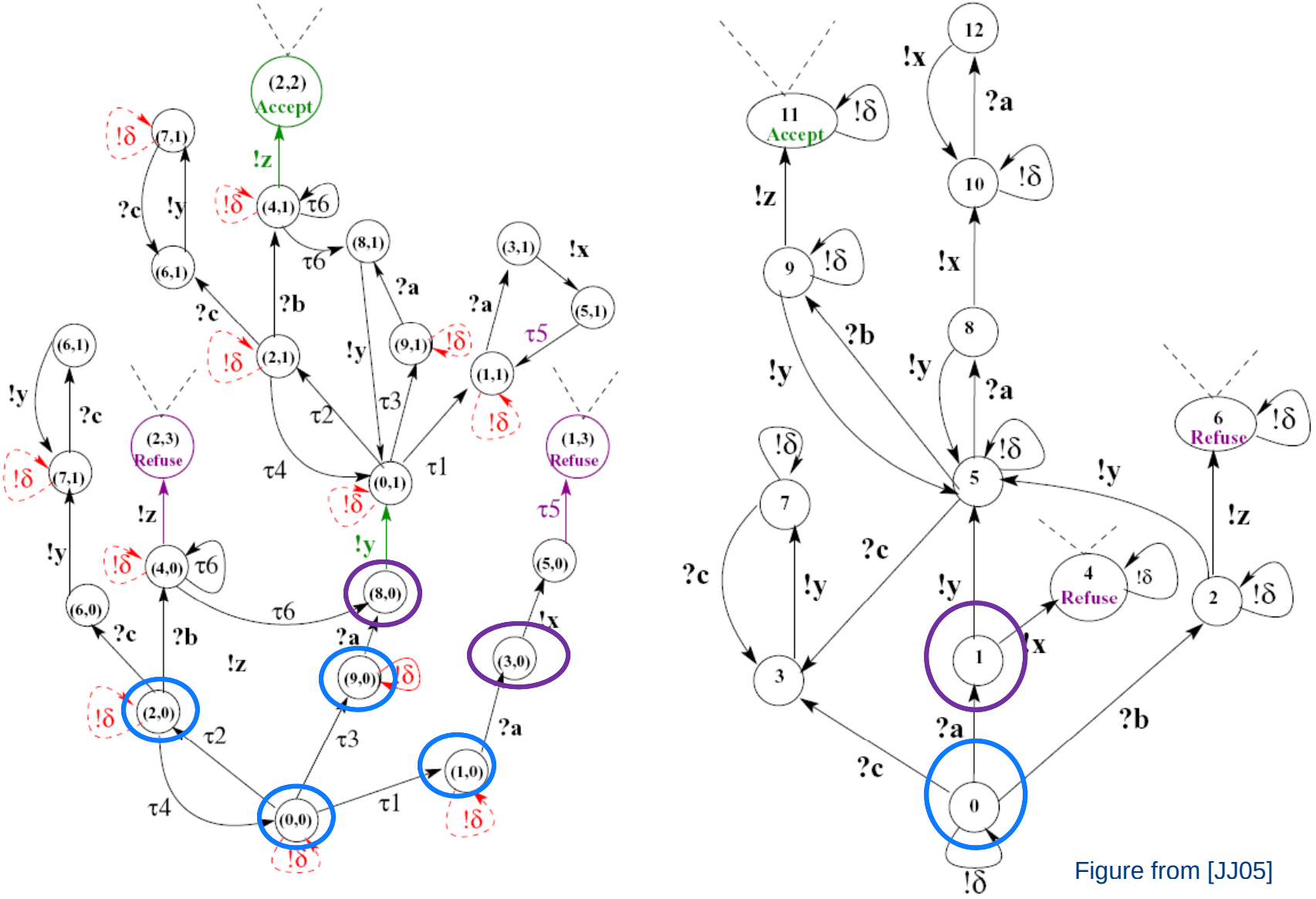
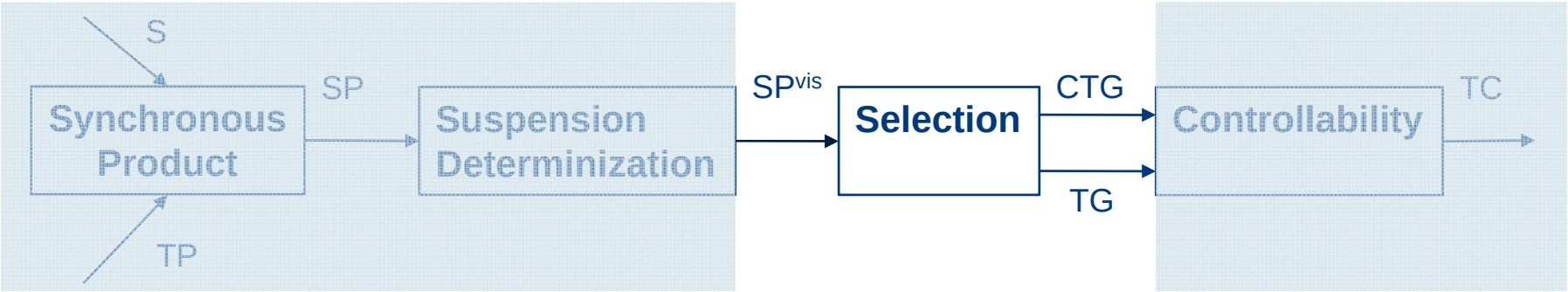


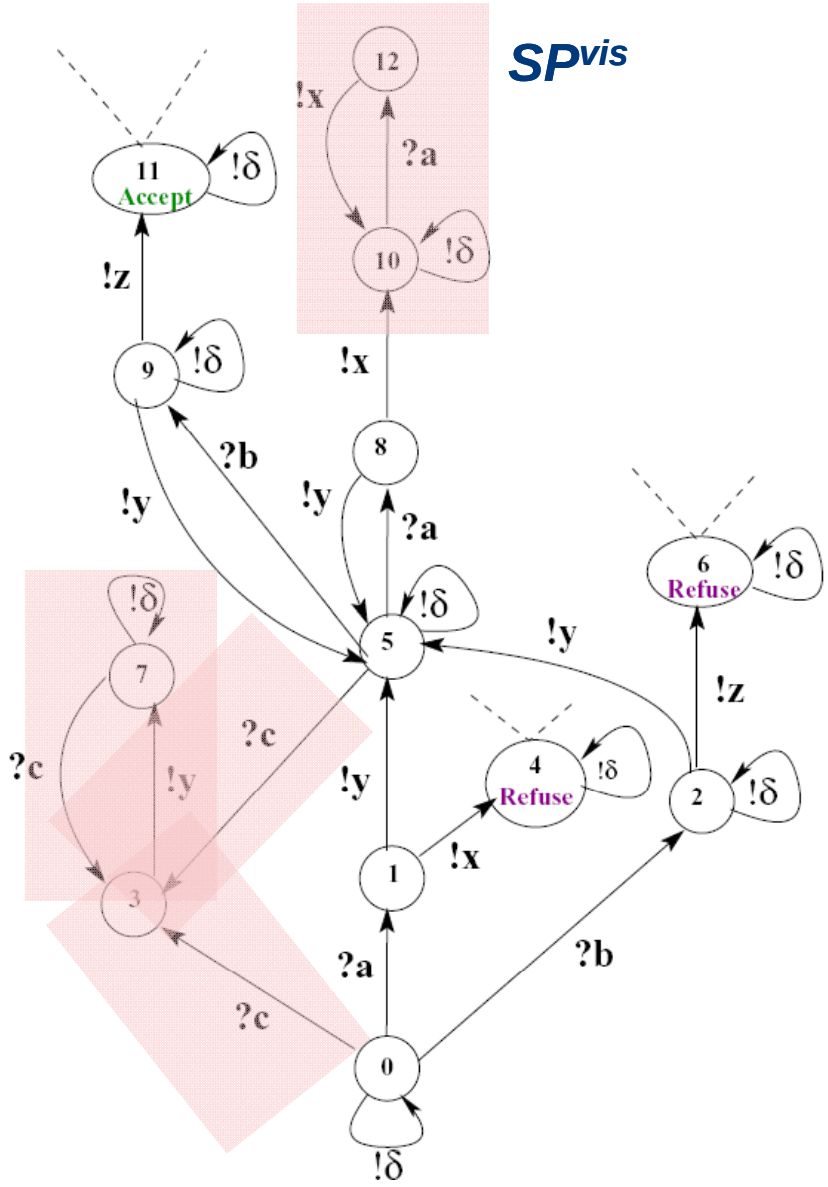
Figure from [JJ05]

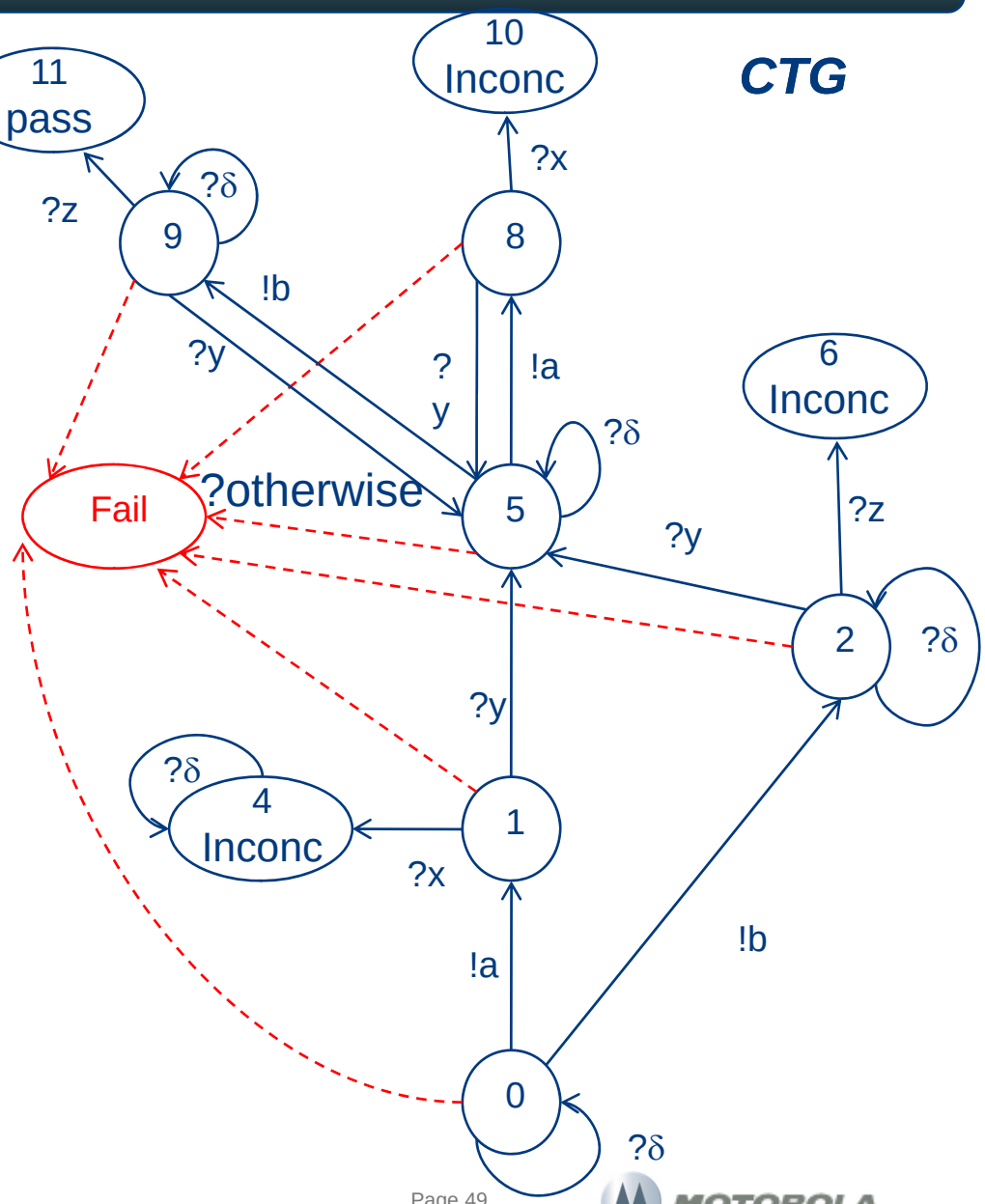
Selection



- Complete Test Graph (CTG) is built by:
  - extracting accepted behaviours;
  - inverting inputs and outputs;
  - sequences that do lead to the accepted or refuse states are removed;
  - adding Pass, Fail and Inconclusive states.
- TG – conflicts can be suppressed during selection

Selection

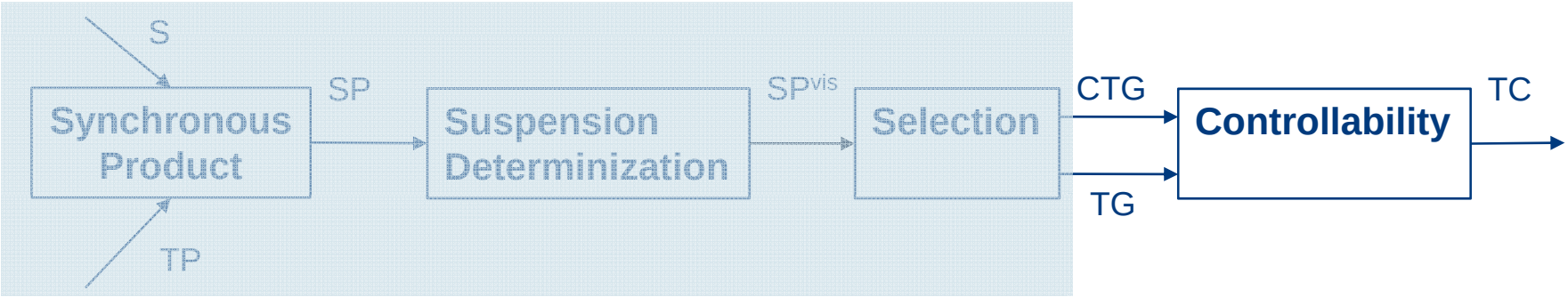




Centro de Informática U·F·P·E

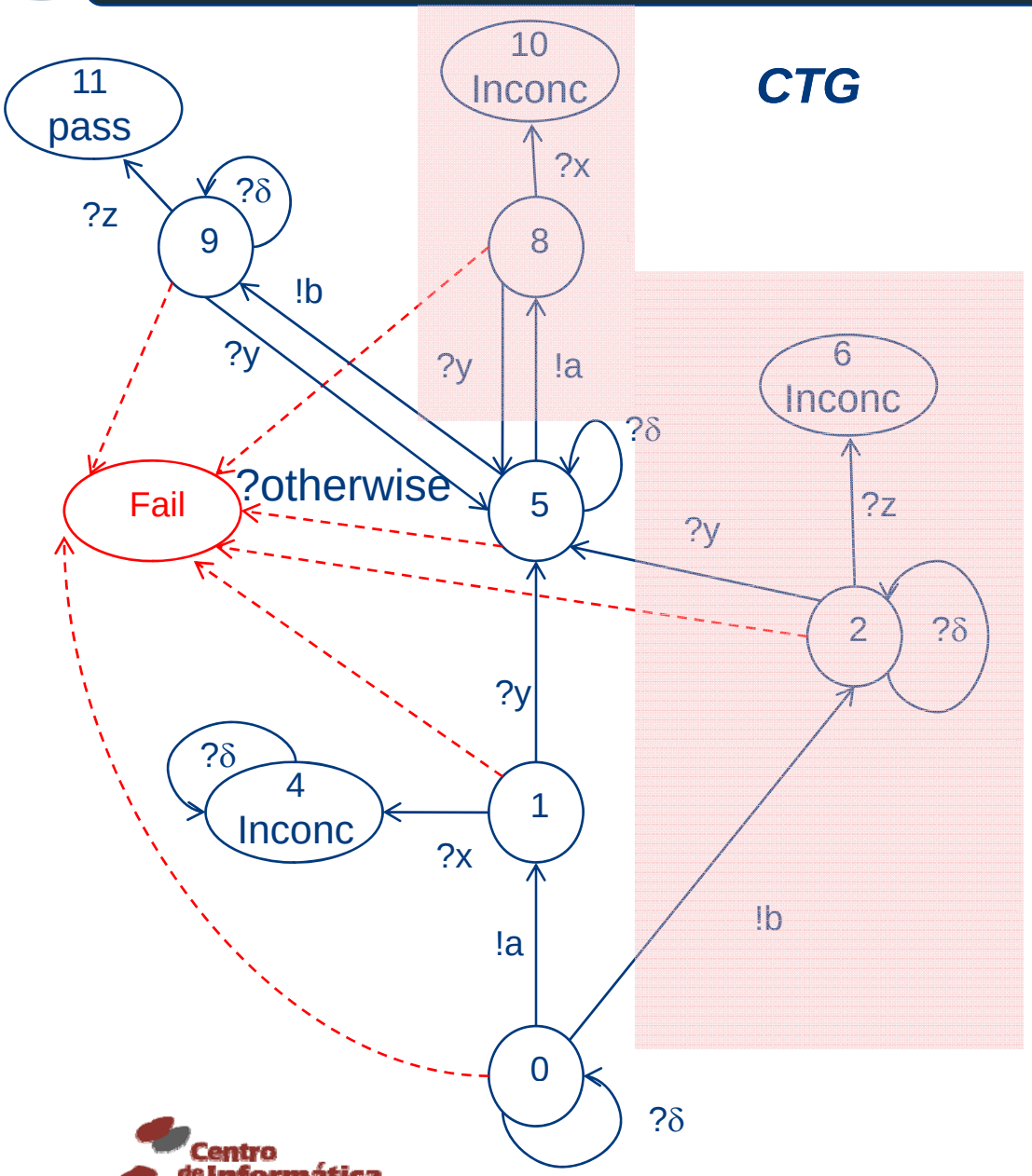
MOTOROLA intelligence everywhere
Mobile Devices Software Group

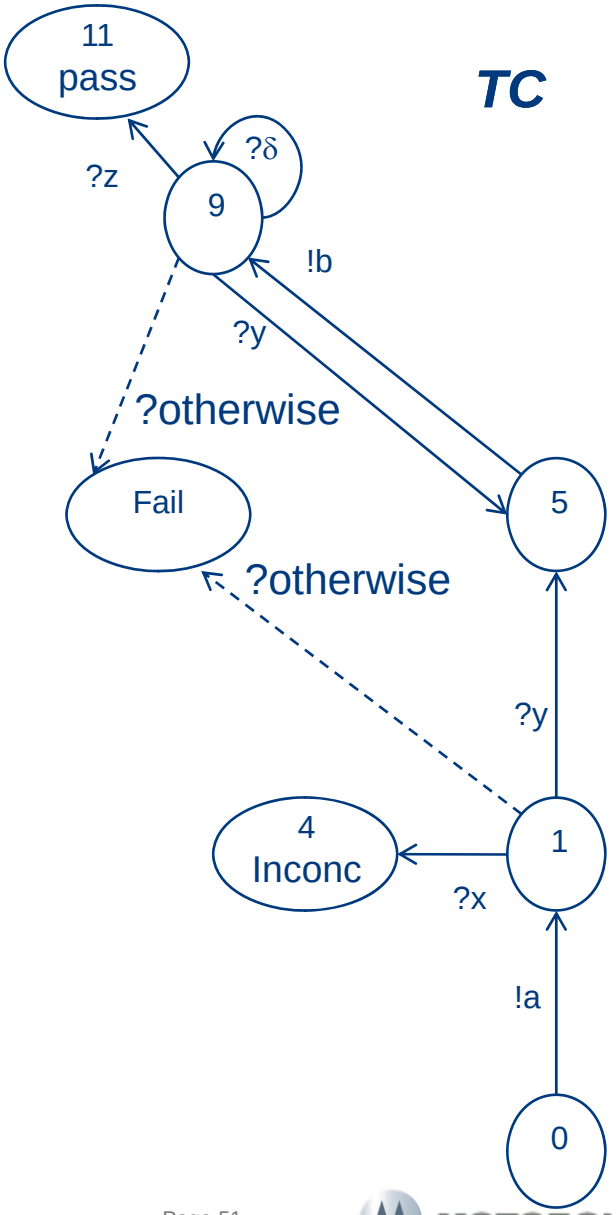
Controllability



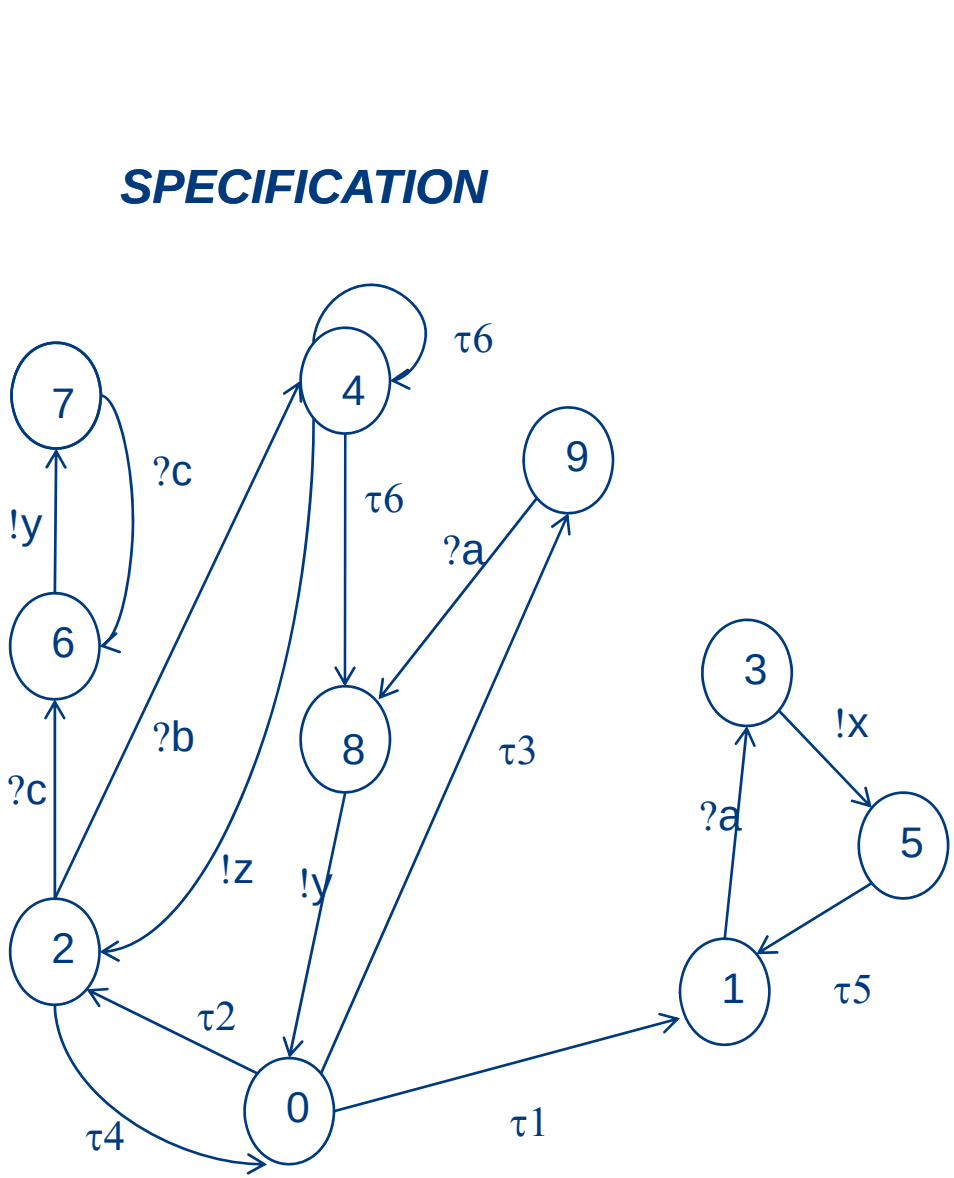
- Controllability conflicts are represented by the presence of choice in some states between:
  - outputs
  - between inputs and outputs
- In a controlled CTG:
  - Either one output is kept and inputs and outputs pruned
  - or all inputs are kept and outputs are pruned

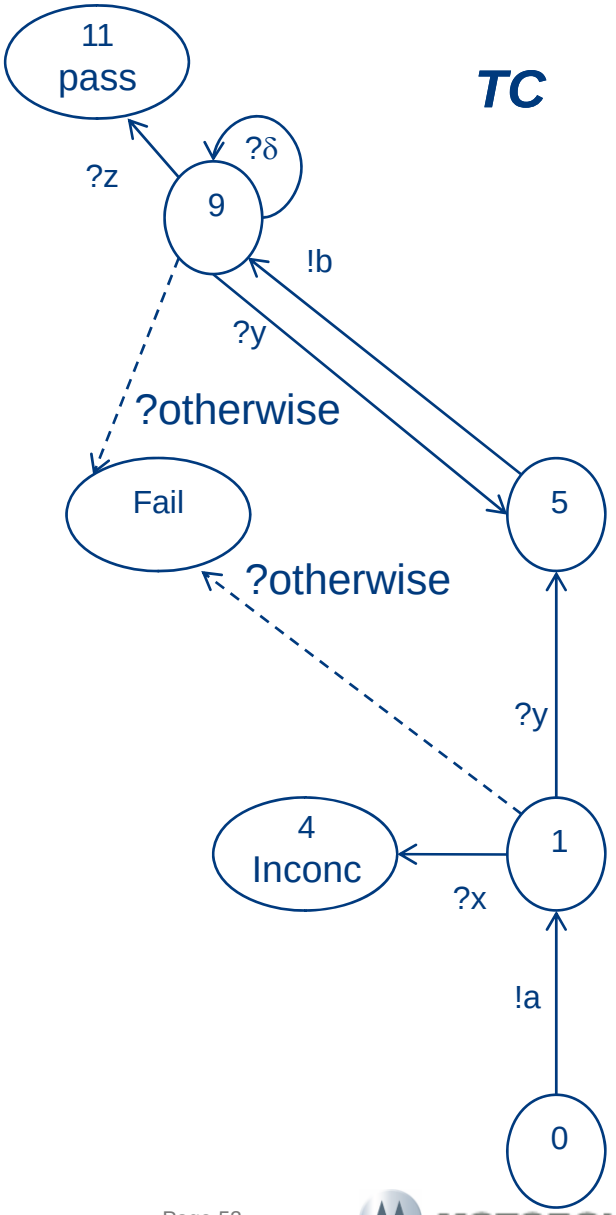
Selection





Soundness and Exhaustiveness





Final Remarks

- TGV test cases are sound and exhaustive;
- Test models can be specified in SDL, Lotos, UML, IF and aldebaram;
- Test Purposes can be specified in aldebaran and UML sequence diagrams;
- Abstract test cases are generated in albebaran and TTCN;
- CADP toolbox.

TEST MODELS AND TEST GENERATION

- Input-Output Labelled Transition Systems
- Annotated Labelled Transition Systems
- Process Algebra
- Markov Chains

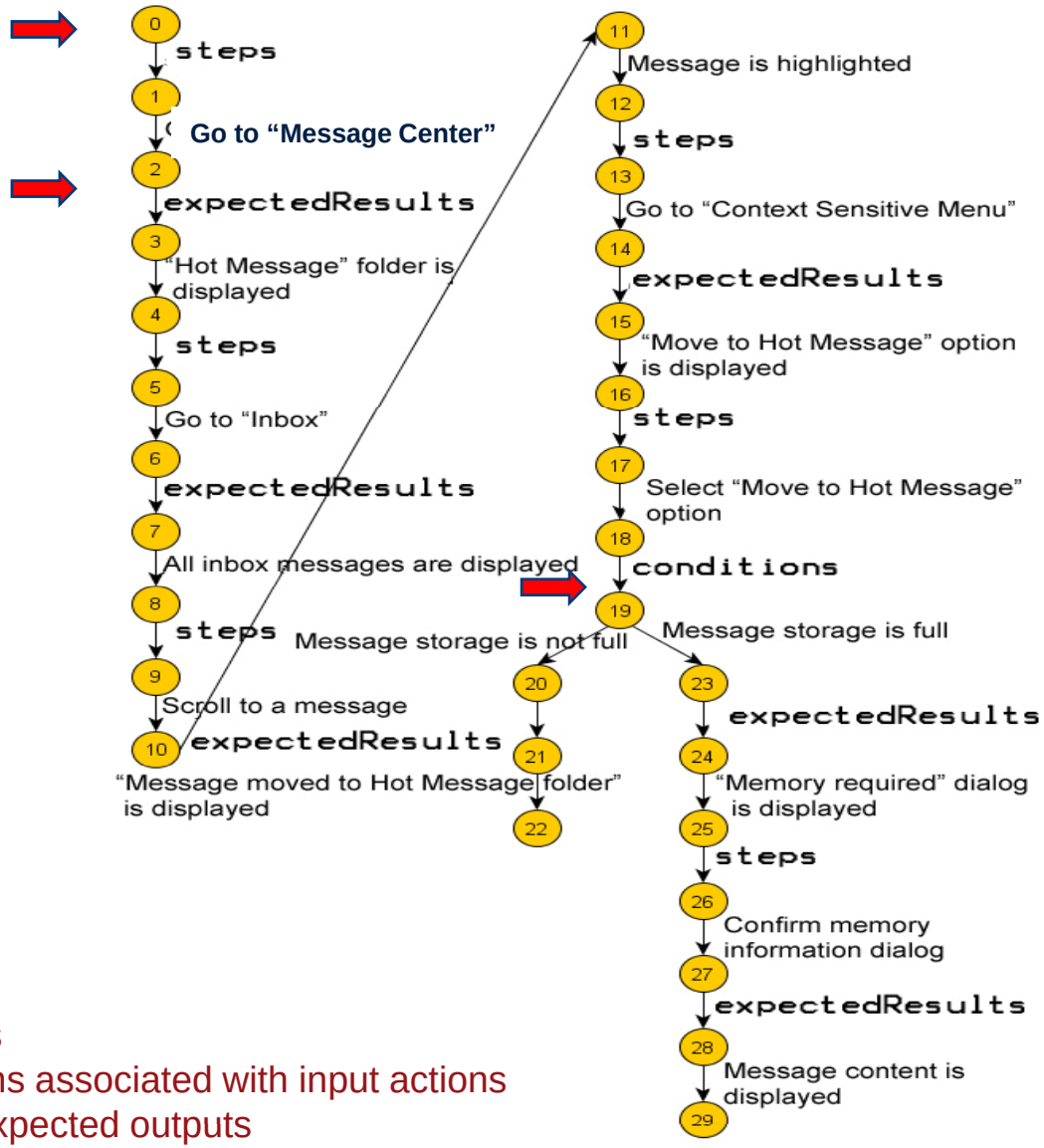
Annotated Labelled Transition System (ALTS)

- Using LTSs for representing feature and interaction behaviour;
- An ALTS is a 4-tuple $M = \langle Q, R, T, q_0 \rangle$, where:
  - Q is a countable, non-empty set of states;
  - $R = A \cup N$ is a countable set of labels, where A is a countable set of actions and N is a countable set of annotations;
  - $T \subseteq Q \times R \times Q$ is a transition relation;
  - $q_0 \in Q$ is an initial state.

Annotated Labelled Transition Systems (ALTS)

- Annotations are inserted into the LTS to:
  - Guide the test case generation through specific interruptions;
  - Make it easier for interruptions models to be composed without interfering with the main model;
  - Guide test case documentation;
  - Make it possible for conditions to be associated with actions;
  - Indicate points where interruptions can be observed.
- ALTS models can be translated into IOLTS models for test case generation using TGV.

Feature ALTS Model



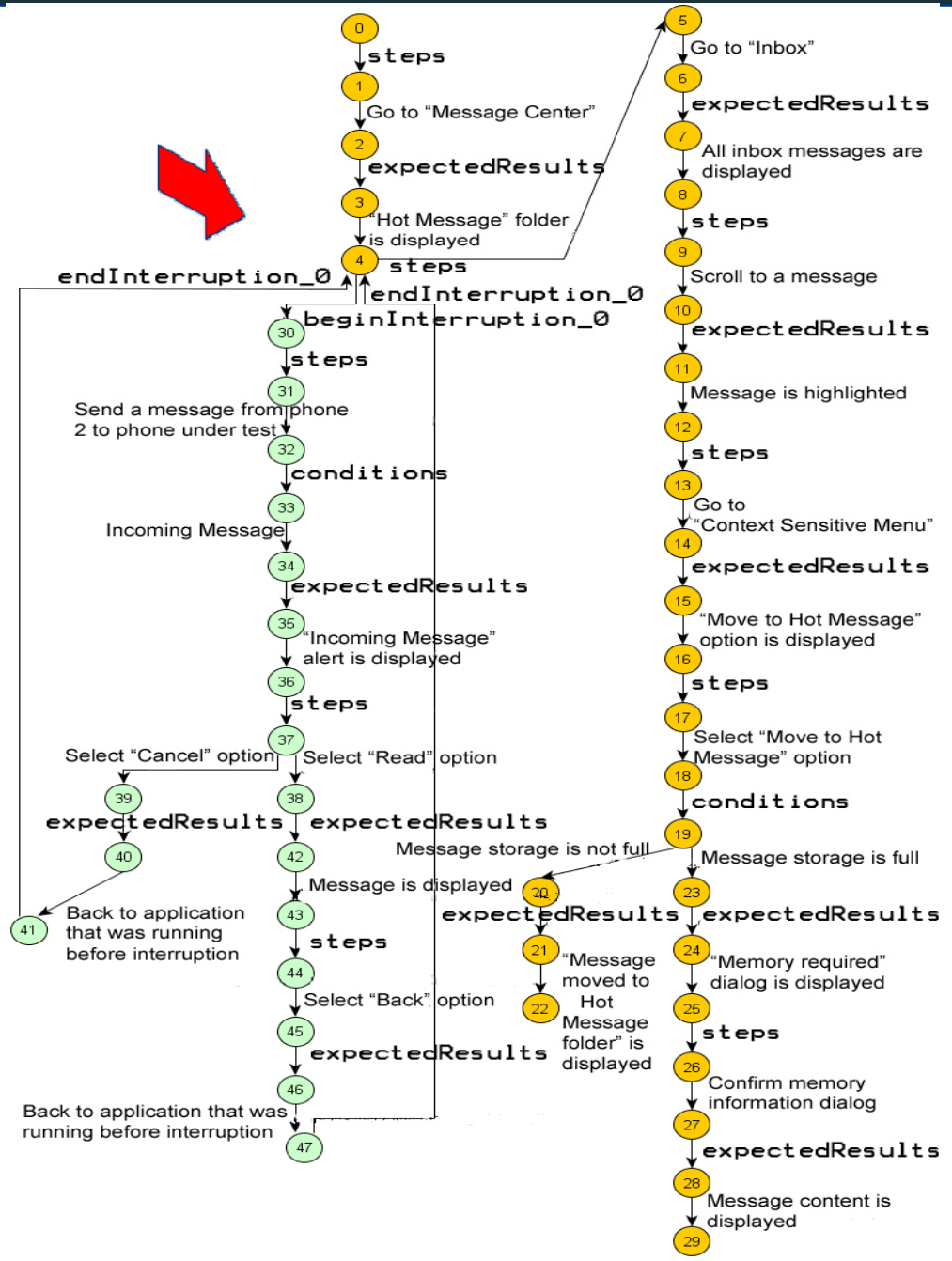
HOT MESSAGE FEATURE

steps → Input Actions
conditions → conditions associated with input actions
expectedResults → Expected outputs

Interruption Testing

- One feature can interrupt the flow of execution of another feature that is running in foreground;
- As a result, features execution are intermingled (combination of independent behaviours);
- An ALTS model with interruption can be defined as follows:
  - The interruption model is plugged in the main feature by adding the *beginInterruption* transition from a target state of an output (expected result);
  - Each leaf node of the interruption model is also plugged in the same state by adding the *endInterruption* annotation;
  - States are re-numbered.

ALTS Model with Interruption



HOT MESSAGE FEATURE interrupted by INCOMING MESSAGE FEATURE

Why interruptions should only be modelled after expected results?

Centro de Informática U·F·P·E

MOTOROLA intelligence everywhere
Mobile Devices Software Group

A Test Case Generation Algorithm (Depth First Search)

```
Decompose (vertex, path, interruptionModel) {
  if (vertex.isLeaf OR (vertex.isRoot AND path <> ∅)) {
            //End of a path
          recordTestCase(path);
          return;
  }
  for each descendent in vertex.getAdjacencies {
          edge ← getEdgeBetween(vertex, descendent);
          if (edge not in path OR edge in interruptionModel.getEdges) {
                  path.add(edge);
                  Decompose(descendent, path, interruptionModel)
          } else recordTestCase(path);
  return
}
```

A Test Case Generation Algorithm (Depth First Search)

```
Decompose (vertex, path, interruptionModel) {
  if (vertex.isLeaf OR (vertex.isRoot AND path <> ∅)) {
            //End of a path
          recordTestCase(path);
          return;
  }
  for each descendent in vertex.getAdjacencies {
          edge ← getEdgeBetween(vertex, descendent);
          if (edge not in path OR edge in interruptionModel.getEdges) {
                  path.add(edge);
                  Decompose(descendent, path, interruptionModel)
          } else recordTestCase(path);
  return
}
```

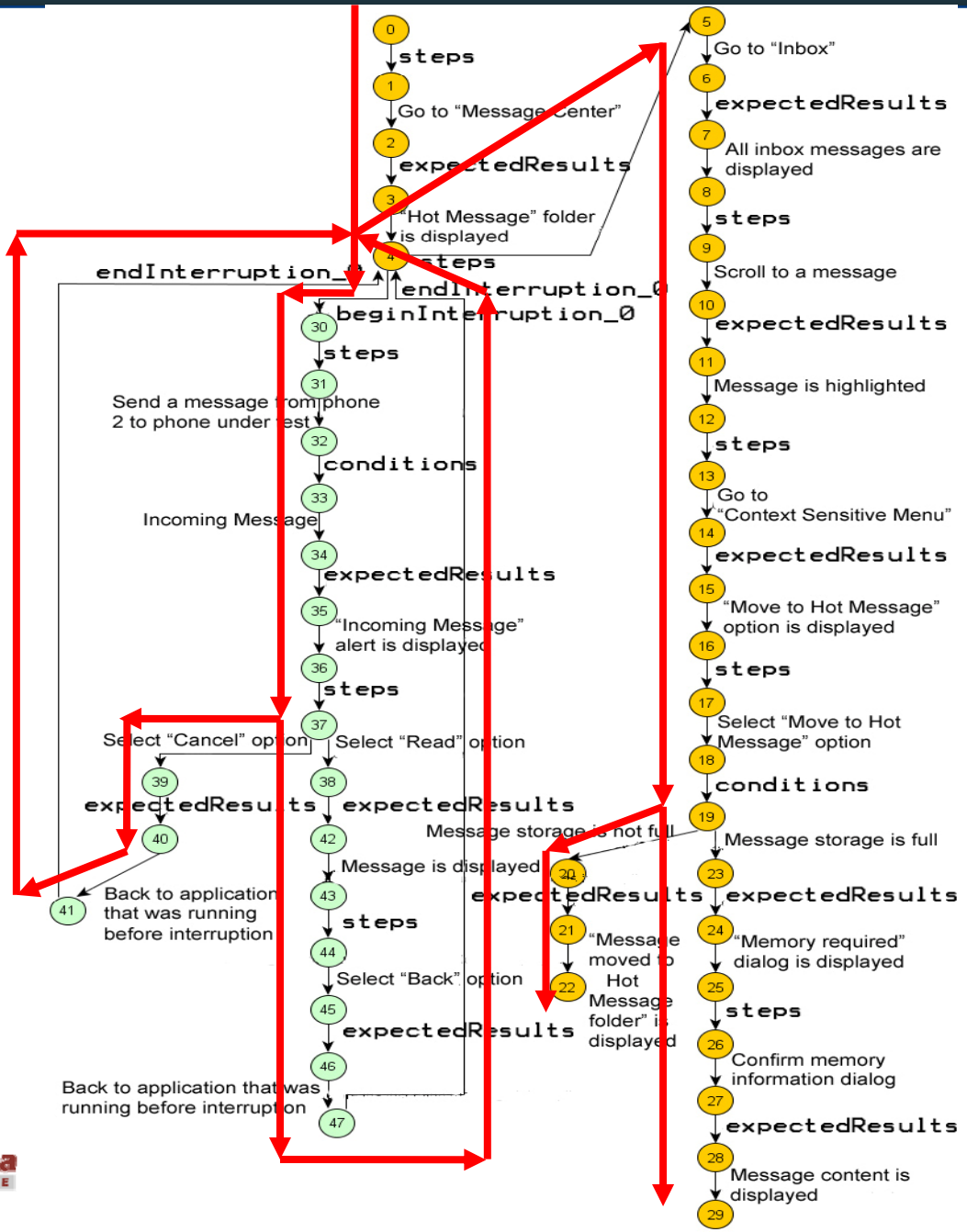
A Test Case Generation Algorithm (Depth First Search)

```
Decompose (vertex, path, interruptionModel) {
  if (vertex.isLeaf OR (vertex.isRoot AND path <> ∅)) {
            //End of a path
          recordTestCase(path);
          return;
  }
  for each descendent in vertex.getAdjacencies {
          edge ← getEdgeBetween(vertex, descendent);
          if (edge not in path OR edge in interruptionModel.getEdges) {
                  path.add(edge);
                  Decompose(descendent, path, interruptionModel)
          } else recordTestCase(path);
  return
}
```

A Test Case Generation Algorithm (Depth First Search)

```
Decompose (vertex, path, interruptionModel) {
  if (vertex.isLeaf OR (vertex.isRoot AND path <> ∅)) {
            //End of a path
          recordTestCase(path);
          return;
  }
  for each descendent in vertex.getAdjacencies {
          edge ← getEdgeBetween(vertex, descendent);
          if (edge not in path OR edge in interruptionModel.getEdges) {
                  path.add(edge);
                  Decompose(descendent, path, interruptionModel)
          } else recordTestCase(path);
  return
}
```

Test Case Generation

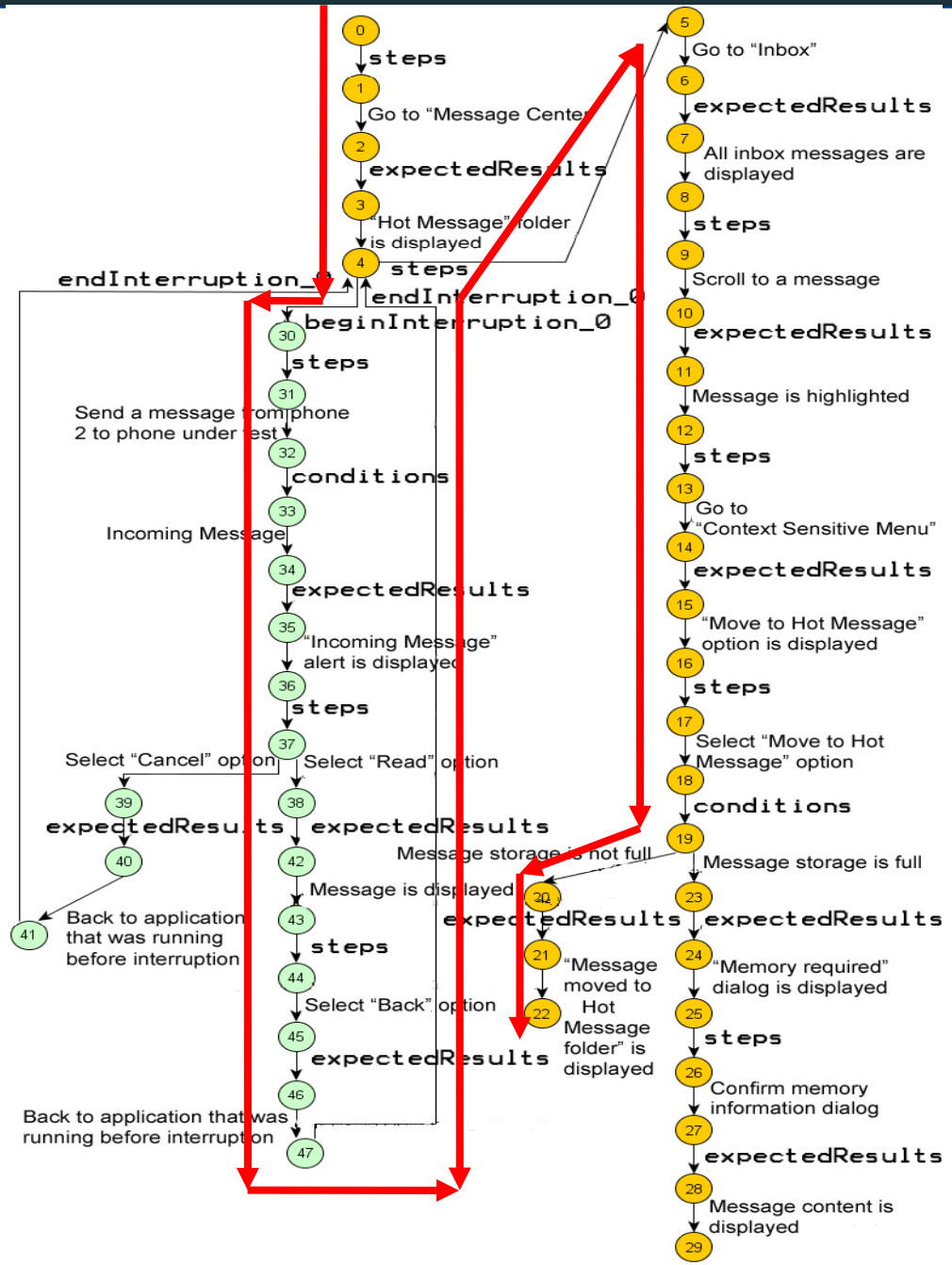


Centro de Informática U·F·P·E

MOTOROLA intelligence everywhere

Mobile Devices Software Group

Test Case Generation

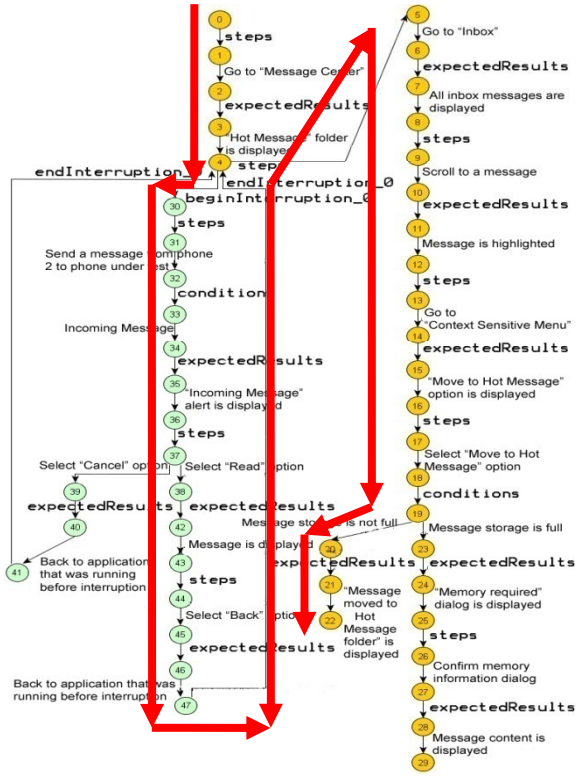


Centro de Informática U·F·P·E

MOTOROLA intelligence everywhere

Mobile Devices Software Group

Interruption Test Case



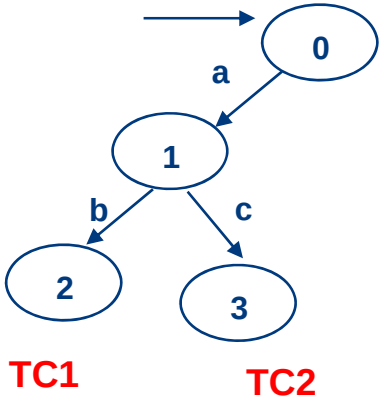
Initial Condition	Message storage is not full
Steps	Expected Results
Go to "Message Center"	"Hot Message" folder is displayed
Send a message from phone 2 to phone under test	"Incoming Message" alert is displayed
Select "Read" option	Message is displayed
Select "Back" button	Back to application that was launching before the incoming message interruption
Go to "Inbox"	All inbox messages are displayed
Scroll to a message	Message is highlighted
Go to "Context Sensitive Menu"	"Move to Hot Message" option is displayed
Select "Move to Hot Message" option	"Message moved to Hot Message folder" is displayed

Test Selection Based on Similarity

- Automatic test case selection from ìnfeasible test suites;
  - The goal is to minimise redundancy of test cases that may be caused by “exhaustive” test case generation;
  - Similar test cases are eliminated until the desired number of test cases is achieved;
- Our Hypothesis:
  - Similar test cases cover a common set of requirements and features and have similar capability of revealing faults;
  - Therefore, some of them can be eliminate to meet resources constraints of a project;
  - There is no additional gain to keep them since they are not significantly affecting requirements/fault model coverage.

Strategy - Similarity-Based Test Case Selection

- Applicable to LTSs in general.
- The Main Steps are:
  1. DFS
     - Test Cases
  2. Calculate degree of similarity between each pair of test cases
     - number of identical transitions
       - "from" and "to" states and transition label are the same for two test cases.
  3. Calculate the length of each test case



Calculating the Degree of Similarity

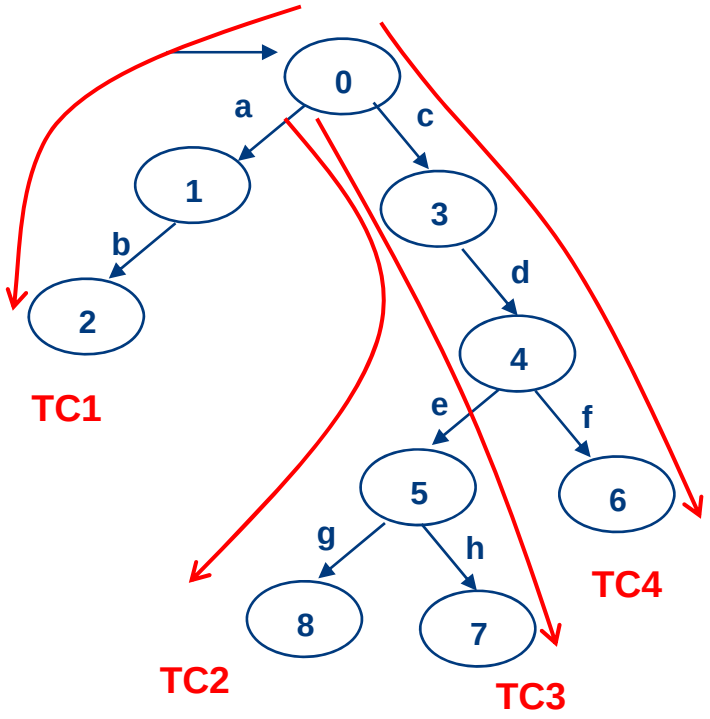
- The degree of similarity between each pair of test cases is calculated and recorded in a **similarity paths matrix.**

- This matrix is:
  - $n \times n$, where n is the number of paths and each n represents one path;
  - Each element a_{ij} is defined by the similarity function applied to a pair of paths.
    - Similarity(i,j) = ***nit / (avg(\|i\|, \|j\|))***, where:
      - *nit* is the number of identical transitions between path *i* and path *j*
      - avg(|*i*|, |*j*|) is the average between paths length

Selecting Test Cases

- By observing the similarity paths matrix, we can define which are the test cases to be excluded:
  - The test cases to be excluded are those that have the highest similarity degree w.r.t. other test cases.
    - The choice is made by observing the test case that has the smallest size.
    - If the size of test cases are same
      - Apply random choice
  - The process is iterative:
    - A test case is removed;
    - The matrix is updated;

Applying the Strategy



|TC1| = 2

|TC2| = 4

|TC3| = 4

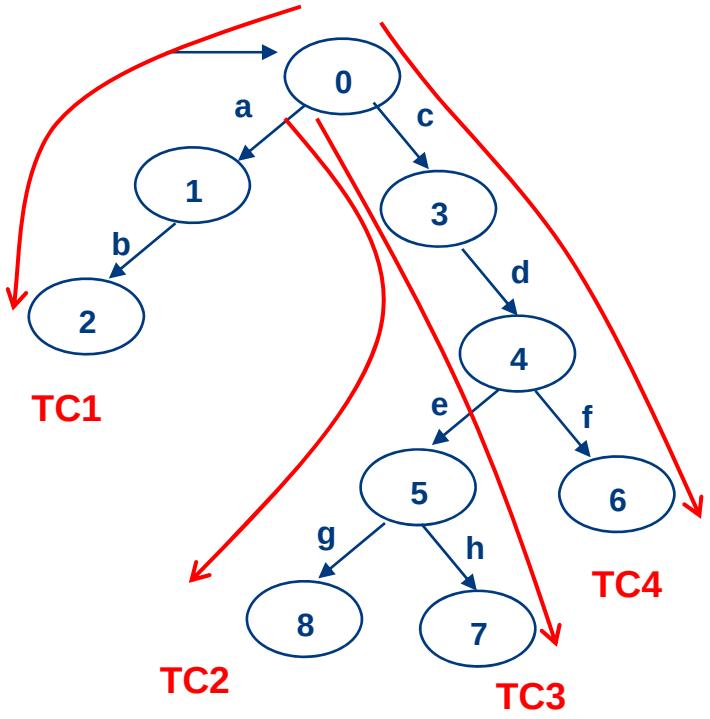
|TC4| = 3

0.75 = 3/ avg(4+4)

	TC1	TC2	TC3	TC4
TC1		0	0	0
TC2			0.75	0.57
TC3				0.57
TC4				

Applying the Strategy

Path coverage: 50% → exclude 2 test cases



|TC1| = 2

|TC2| = 4

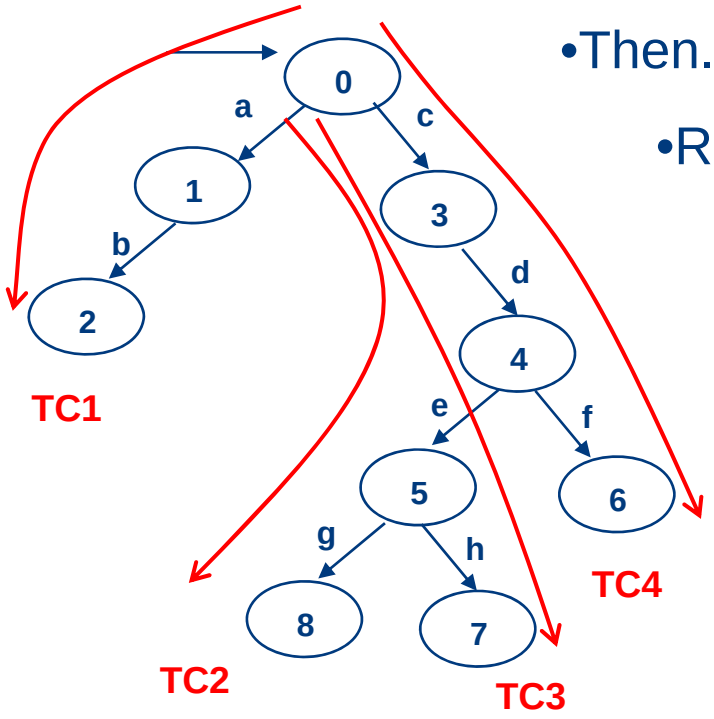
|TC3| = 4

|TC4| = 3

	TC1	TC2	TC3	TC4
TC1		0	0	0
TC2			0.75	0.57
TC3				0.57
TC4				

Applying the Strategy

- TC2 or TC3?
- By size?
  - |TC2| = |TC3| = 4
  - NO
- Then..
  - Random (TC2,TC3) = TC2



|TC1| = 2

|TC2| = 4

|TC3| = 4

|TC4| = 3

	TC1	TC2	TC3	TC4
TC1		0	0	0
TC2			0,75	0,57
TC3				0,57
TC4				

Applying the Strategy

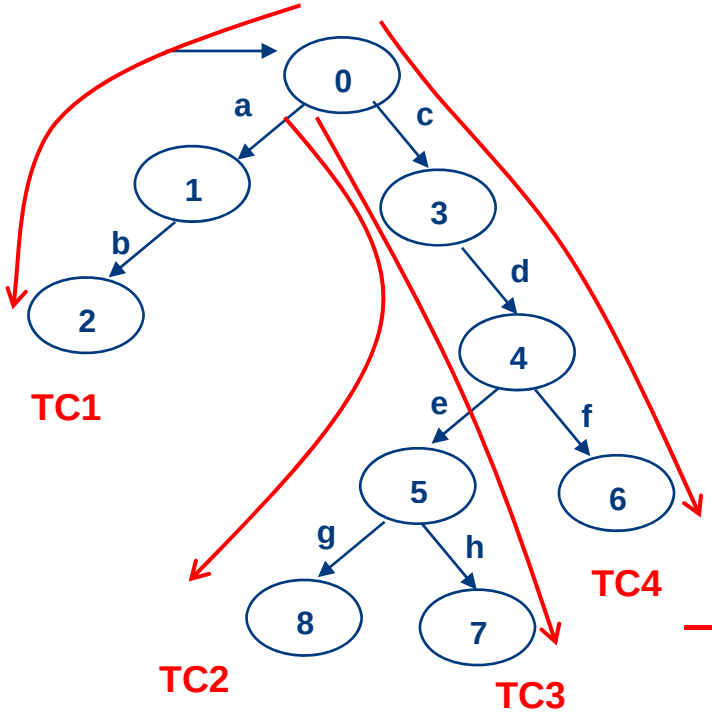
- Random (TC2,TC3) = TC2

|TC1| = 2

|TC2| = 4

|TC3| = 4

|TC4| = 3



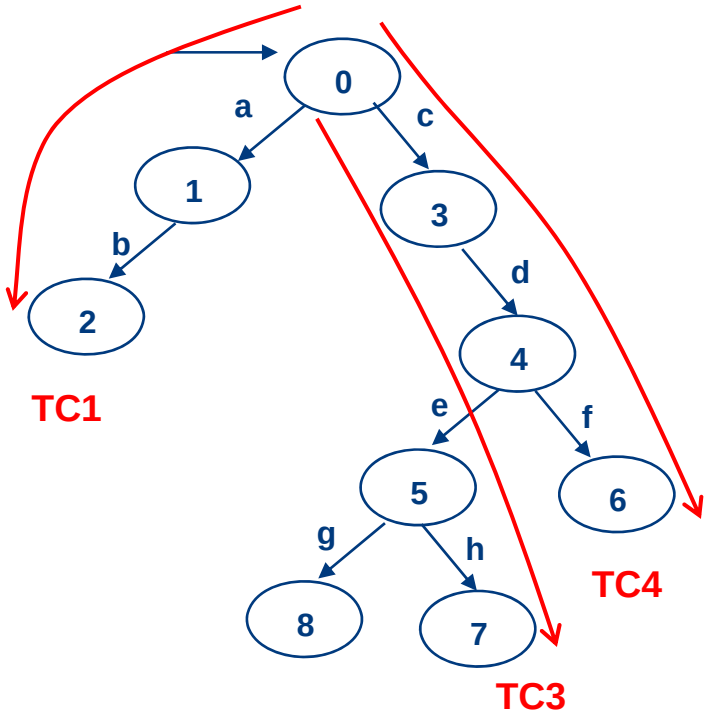
	TC1	TC2	TC3	TC4
TC1		0	0	0
TC2			0,75	0,57
TC3				0,57
TC4				

Applying the Strategy

|TC1| = 2

|TC3| = 4

|TC4| = 3



	TC1	TC3	TC4
TC1		0	0
TC3			0,57
TC4			

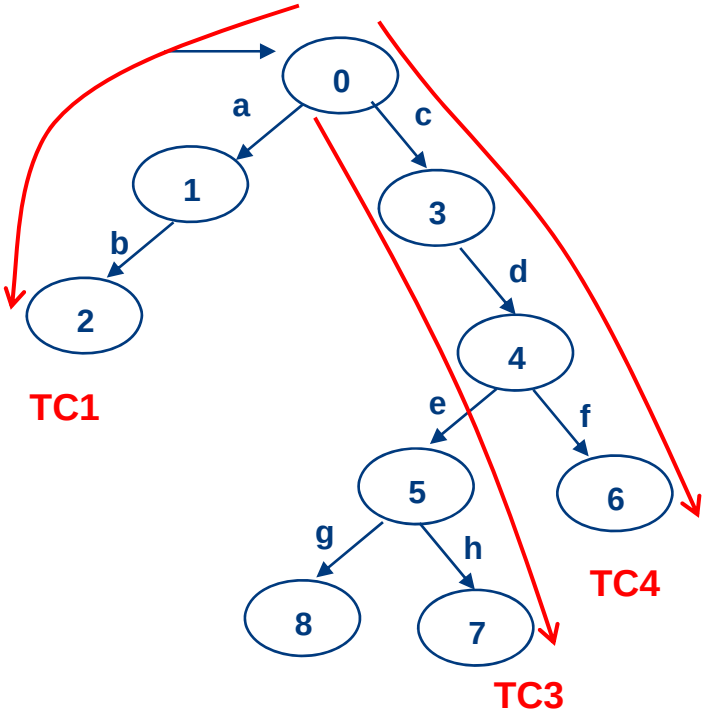
Applying the Strategy

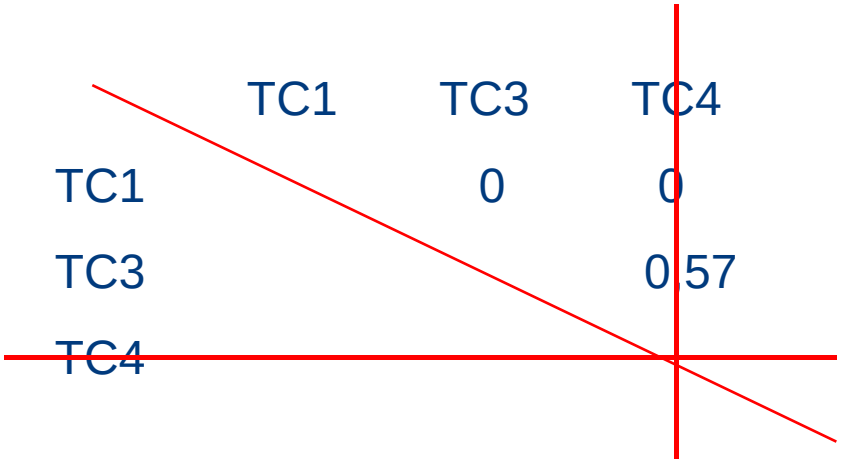
- TC3 or TC4?
- |TC3| > |TC4|

|TC1| = 2

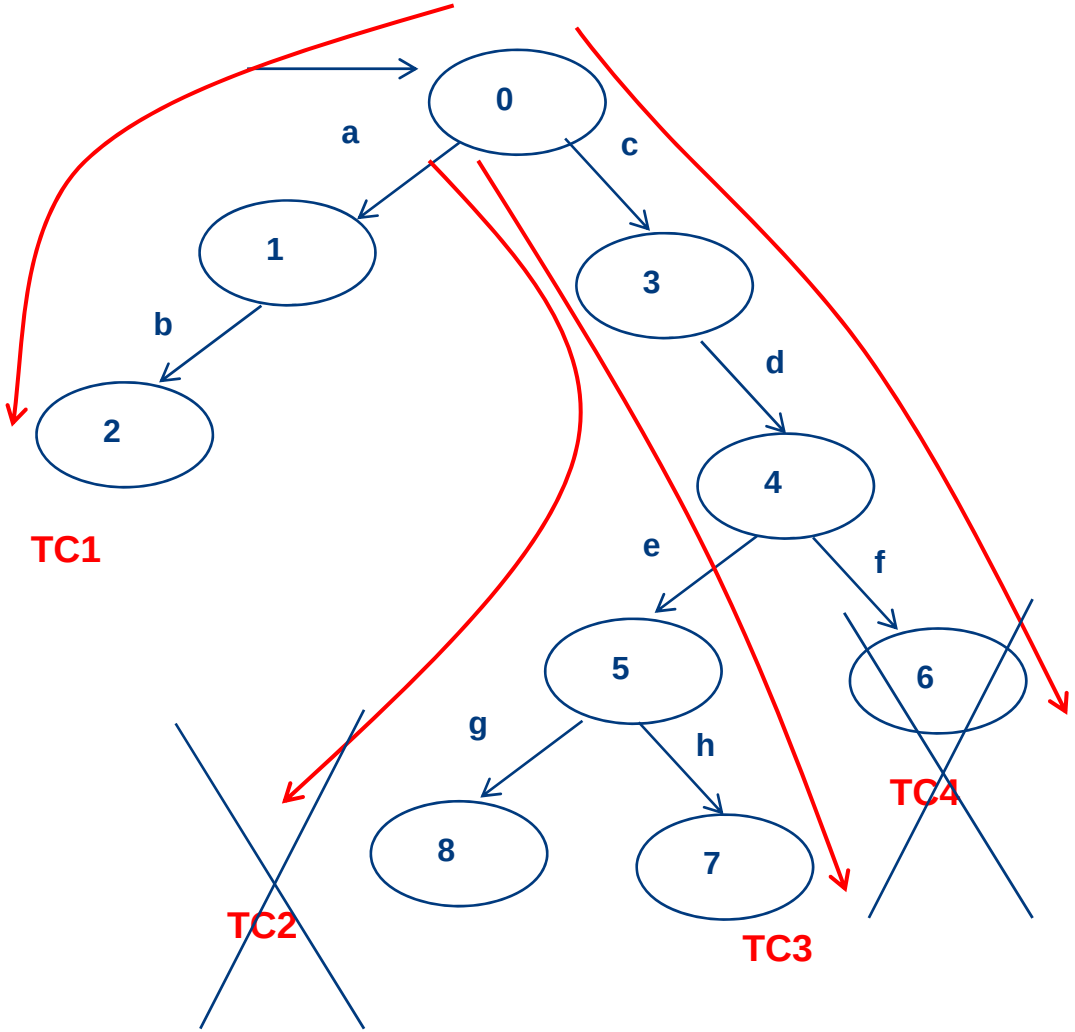
|TC3| = 4

|TC4| = 3





Applying the Strategy



Evaluating the Similarity-Based Test Case Selection Strategy

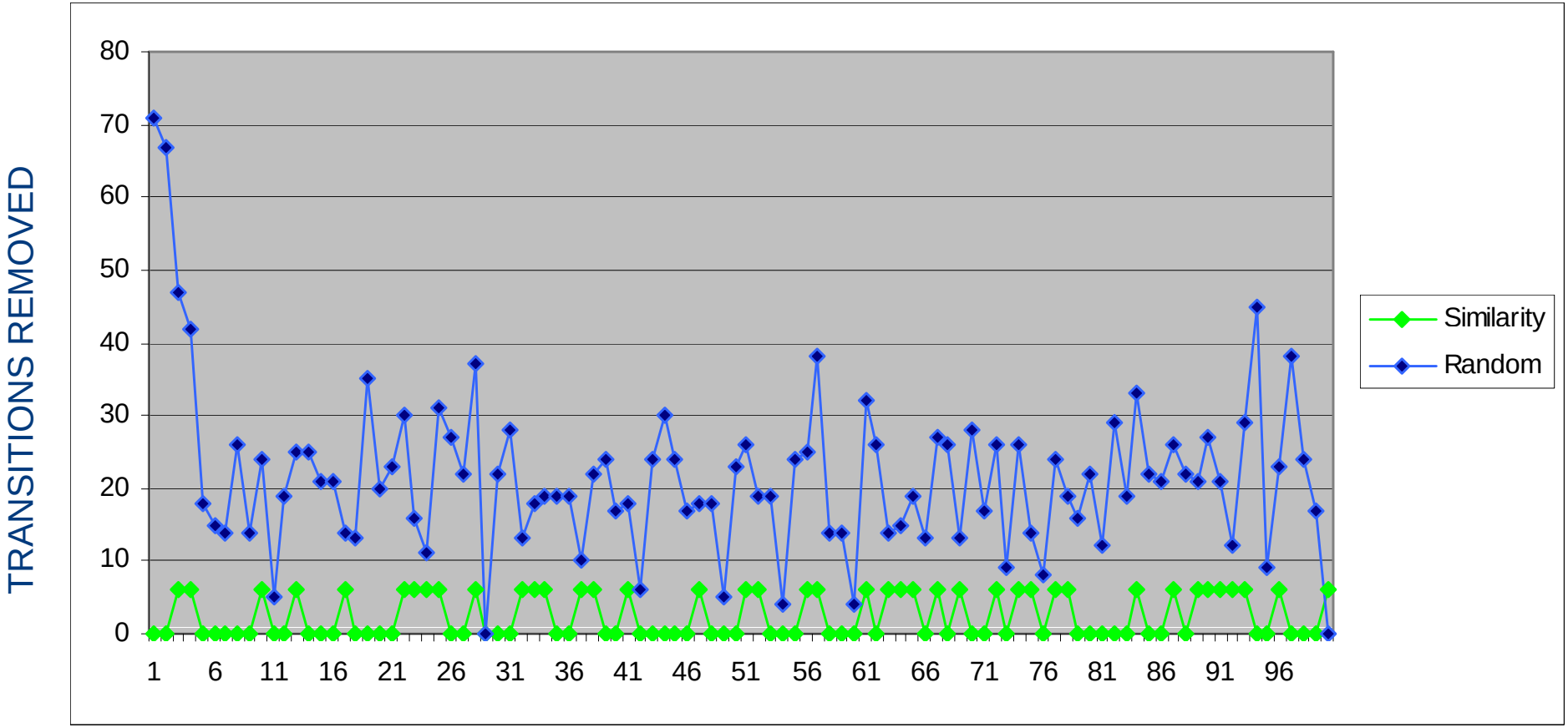
- Our impression from the use of this strategy is that it selects the most different test cases.
  - To assess, we compared our strategy with a random strategy.
- Three different reactive applications were chosen.
- The goal was to measure the percentage of transitions coverage in the two strategies.
- Two criteria were considered
  - 50% path coverage
  - For each application, we applied 100 times our strategy and the random strategy

The Case Studies

- Reactive applications: two mobile phone applications and the Target Tool, modelled according to our methodology (templates + ALTS)
- Case study 1 is a feature for adding contacts in a mobile phone's contact list;
- Case study 2 is a message application that deals with embedded items. An embedded item can be a URL, phone number or e-mail. For each embedded item, it is possible to execute some tasks.
- Case study 3 is the Target tool, an application that generates test cases automatically from use case scenarios.

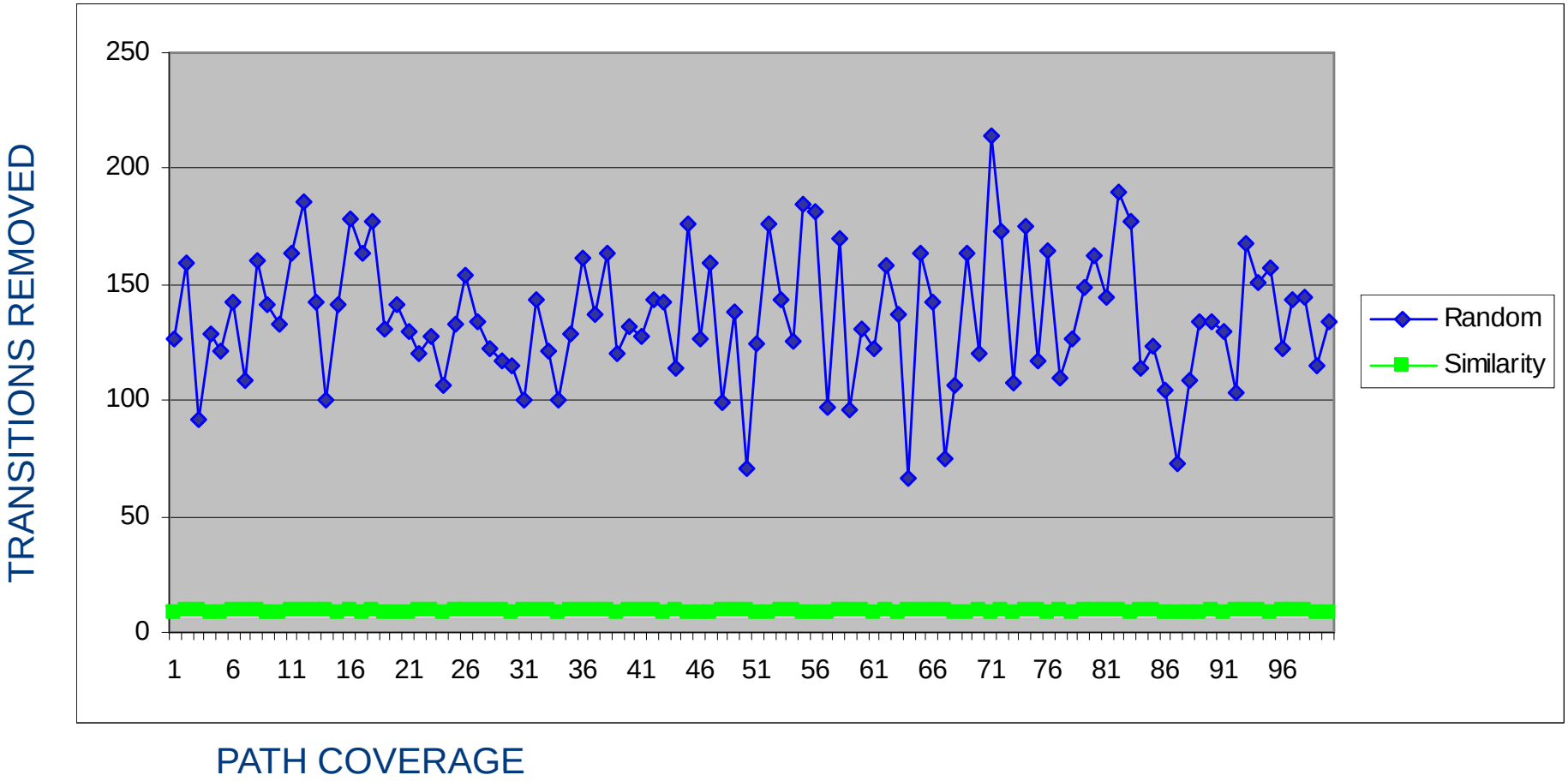
Evaluating the Similarity-Based Test Case Selection Strategy

- Adding contacts in a mobile phone's contact list
- Feature with similar test cases of similar length



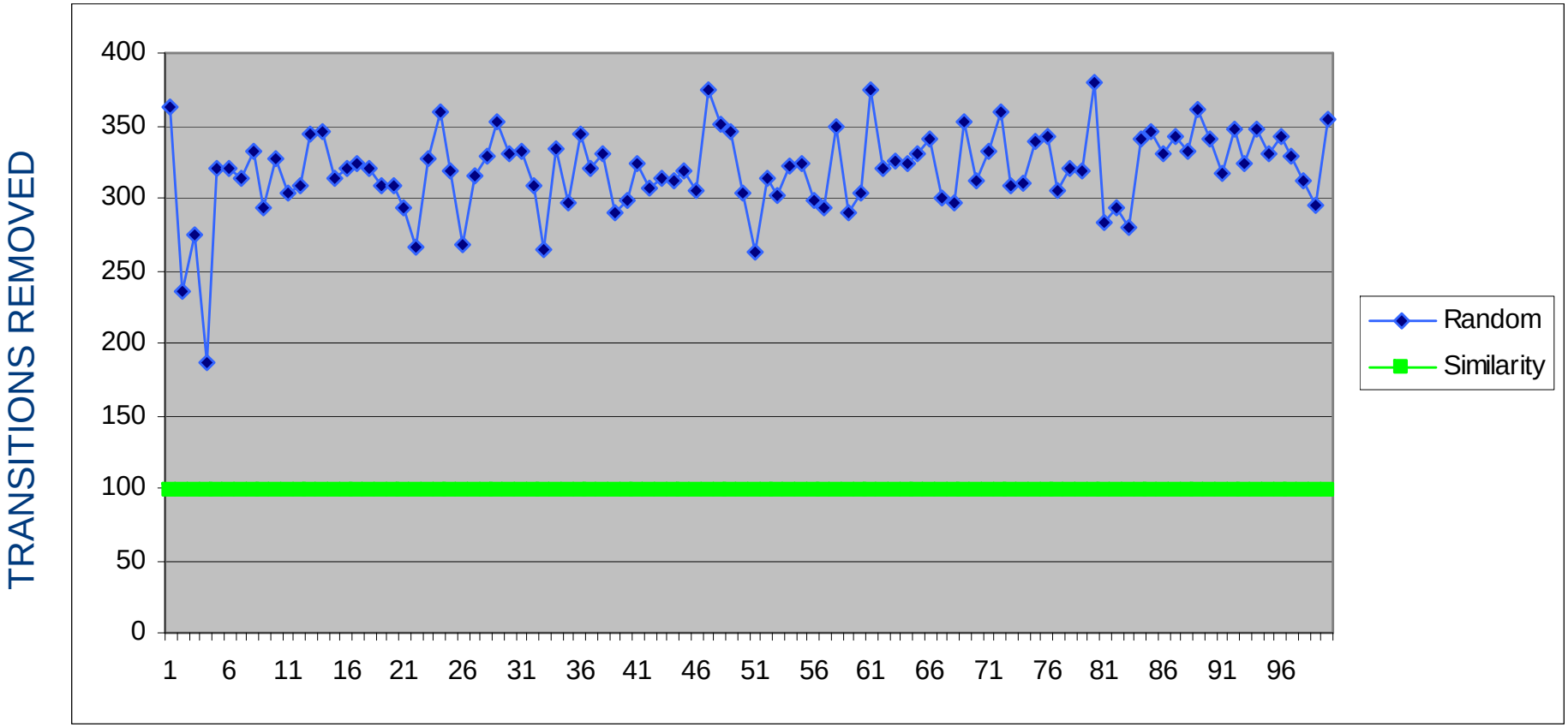
Evaluating the Similarity-Based Test Case Selection Strategy

- Message application that deals with embedded items
- Different features compose this application. Groups of similar test cases



Evaluating the Similarity-Based Test Case Selection Strategy

- The Target tool
- GUI application with different features. Different length of test cases.



Final Considerations

- ALTS models are suitable for feature and interruption test model and test case generation;
- These are abstract models that can be transformed into formal models such as IOLTSs;
- Test generation algorithm generates sound and exhaustive test cases for deterministic models;
- These models can be generated from use case templates with different combinations of allowed interruptions at different points of the execution flow of features;
- Automatic test case selection can be applied to focus test generation and minimize the size of test suites.

TEST MODELS AND TEST GENERATION

- Input-Output Labelled Transition Systems
- Annotated Labelled Transition Systems
- Process Algebra
- Markov Chains

Process algebras

- Abstract and formal treatment of concurrency
- Rich repertoire of process composition operators
- Some examples: CSS, CSP, LOTOS
- A variety of semantic models
  - Operational
  - Denotational
  - Algebraic

Test model and test generation

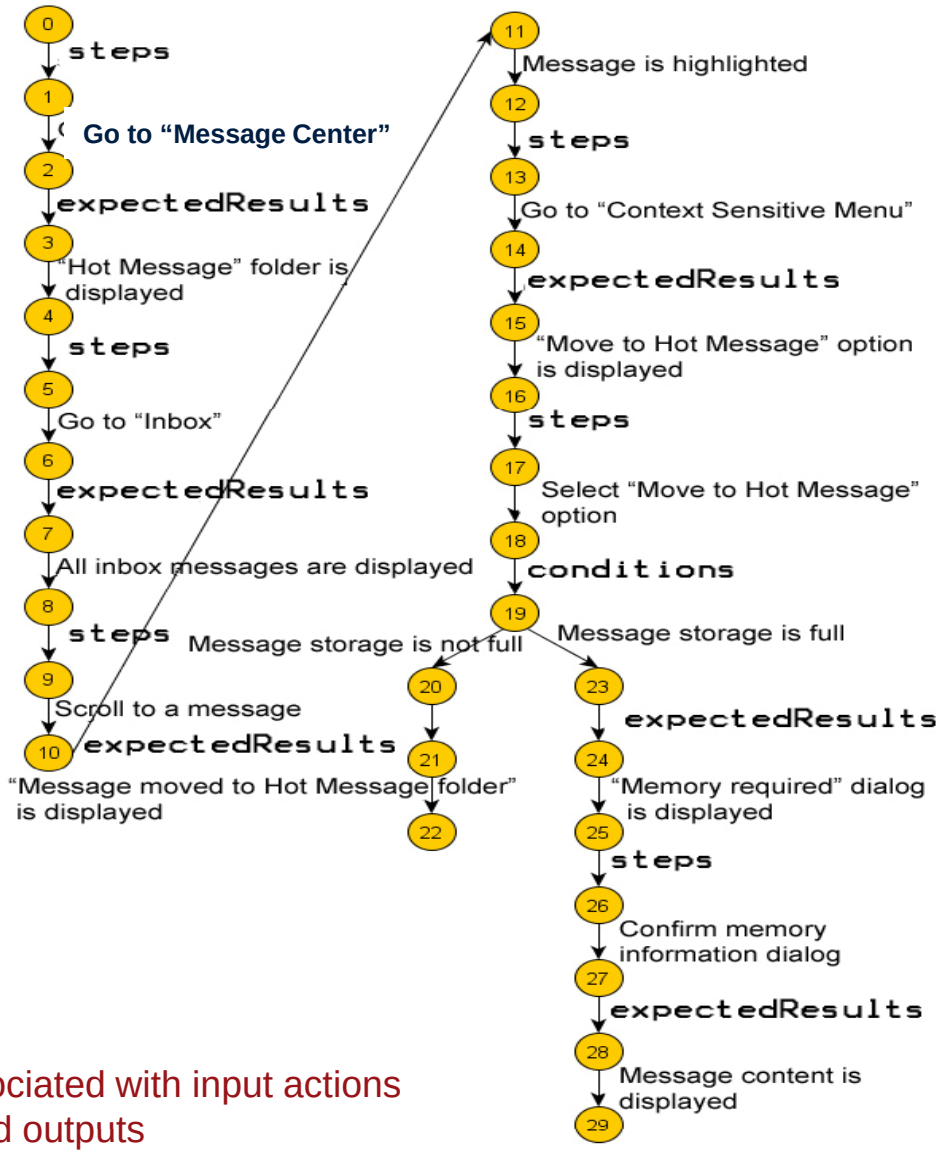
- Abstract test models with rich structure
- Test generation via model checking

  No explicit algorithm
- Test selection via process composition (possibly based on the application architecture)
- Composition of test models, test purposes, test cases …
- Solid framework for formalisation and reasoning

Test model: Communicating Sequential Processes (CSP)

- Specification of concurrent and distributed systems
- Primitive processes: **`SKIP, STOP`**
- Prefix: **`a -> P`**
- Sequential Composition: **`P ; Q`**
- Choice: **`P [] Q`** and **`P |~| Q`**
- Alphabetized parallel: **`P ||[C]|| Q`**
- Interleaving: **`P ||| Q`**
- Hiding: **`P \ C`**

How to describe this ALTS Model in CSP?



HOT MESSAGE FEATURE

steps → Input Actions
conditions → conditions associated with input actions
expectedResults → Expected outputs

Example: Moving a Message from Inbox to the Hot Message Folder

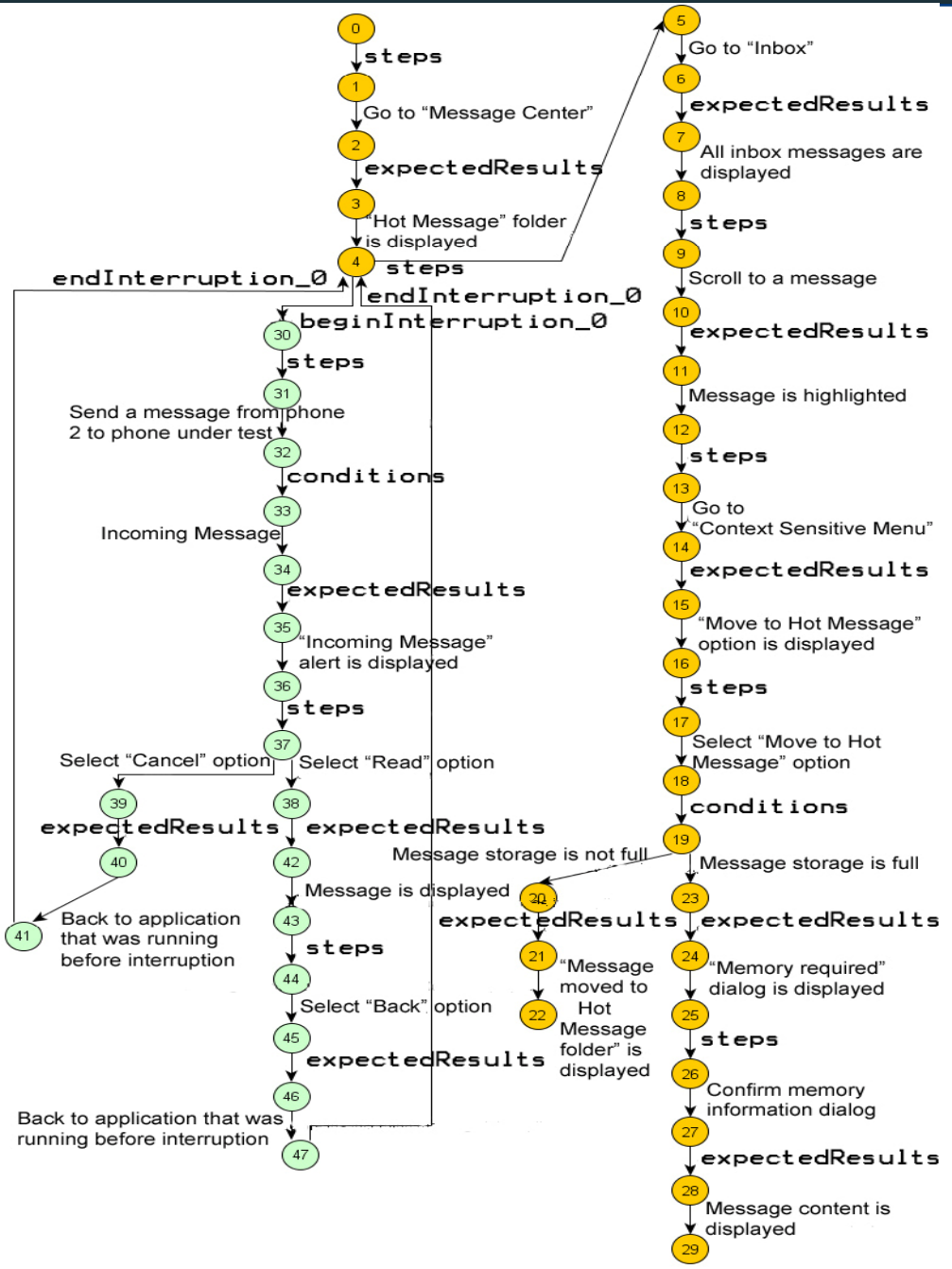
```
UC1 =
 goToMessageCenter ->
     hotMessageFolderIsDisplayed ->
 goToInbox ->
     allInboxMessagesAreDisplayed ->
 scrollToAMessage ->
     messageIsHighlighted ->
 goToContextSensitiveMessage ->
     moveToHotMessagesOptionIsDisplayed ->
 selectMoveToHotMessagesOption -> (UC11 [] UC12)
```

Example: Moving a Message from Inbox to the Hot Message Folder

```
UC11 =
 messageStorageIsNotFull ->
     messageMovedToHotMessageFolderDisplayed ->
 SKIP

UC12 =
 messageStorageIsFull ->
     memoryRequiredDialogIsDisplayed ->
 confirmMemoryInformationDialog ->
     messageContentIsDisplayed ->
 SKIP
```

ALTS Model with Interruption



HOT MESSAGE FEATURE interrupted by INCOMING MESSAGE FEATURE

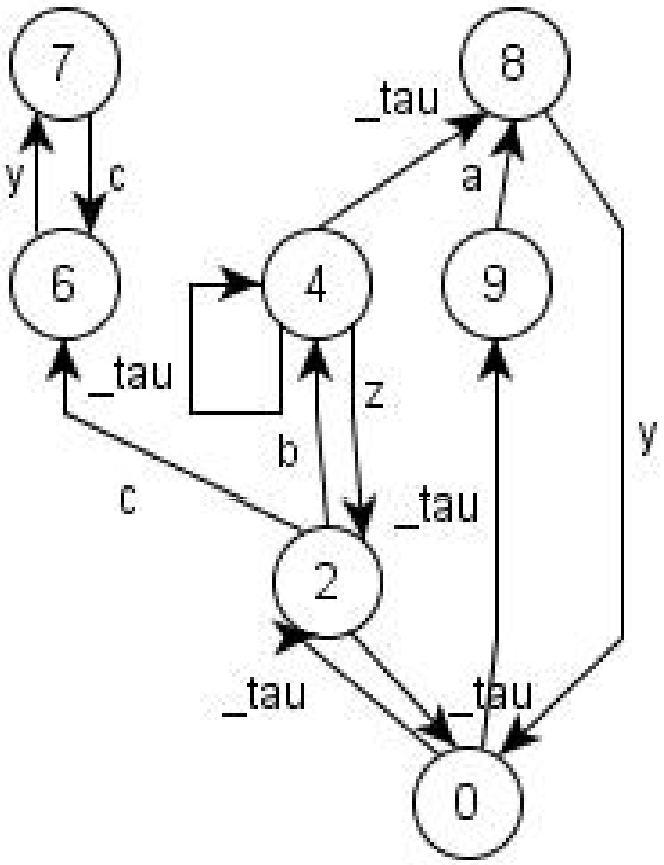
Centro de Informática U·F·P·E

MOTOROLA intelligence everywhere
Mobile Devices Software Group

Capturing interruptions

```
UCI = UC1 []
      (UC1I |[beginI,endI]| INTERRUPTION)\{beginI,endI}
UC1I =
 goToMessageCenter ->
      HotMessageFolderIsDisplayed ->
 beginI -> endI ->
 ...
INTERRUPTION =
 beginI ->
 ...
 endI -> SKIP
```

Distinguishing input and output




```
channel a,b,c,y,z,t

AlfaSi = {a,b,c}

AlfaSo = {y,z}

AlfaS = AlfaSi U AlfaSo
S0 = t -> S9 [] t -> S2
S2 = t -> S0 [] (c -> S6
     [] b -> S4)
S4 = z -> S2 [] t -> S4
     [] t -> S8
S6 = y -> S7
S7 = c -> S6
S8 = y -> S0
S9 = a -> S8
SYSTEM = S0\{t}
```

Traces semantics

$$traces(STOP) = \{\langle\rangle\}$$

$$traces(SKIP) = \{\langle\rangle, \langle\surd\rangle\}$$

$$traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown t \mid t \in traces(P)\}$$

$$traces(P \,[]\, Q) = traces(P \,|\sim|\, Q) = traces(P) \cup traces(Q)$$

$$traces(P;Q) = \{s \mid s \in traces(P) \wedge \neg \langle\surd\rangle \text{ in } s\} \cup \{s \frown t \mid s \frown \langle\surd\rangle \in traces(P) \wedge t \in traces(Q)\}$$

...

Example: $P = (a \rightarrow b \rightarrow STOP) \,[]\, (c \rightarrow SKIP)$

$traces(P) = \{\langle\rangle, \langle a\rangle, \langle a,b\rangle, \langle c\rangle, \langle c,\surd\rangle\}$

Traces refinement

$$P \sqsubseteq_T Q \quad \text{iff} \quad traces(Q) \subseteq traces(P)$$

- Example: **(P [] Q) [T= P**
- Other semantic models consider *failures* and *divergencies* of a process

Test generation from CSP models

- Unlike LTS based approaches, no explicit algorithm is defined
- Generation via (traces) refinement checking
- Test model must be annotated with *marking* events

```
accept, refuse
```

Test generation from CSP models

- Example: marking the **UC1** model

  UC1T = UC1 ; accept -> STOP

- Exercise: expand **UC1T**

- Exercise: relate the traces of **UC1** and **UC1T**

- Exercise: does the refinement

  UC1 [T= UC1T

  hold?

Test generation from CSP models

- The counter examples of the refinement

  `UC1 [T= UC1T`

  are precisely the relevant traces (test scenarios)

- But FDR yields only one counter example

- For our example:

```
ts1 = <goToMessageCenter,
  hotMessageFolderIsDisplayed, ...,
  messageStorageIsNotFull,
  messageMovedToHotMessageFolderDisplayed>
```

Test generation from CSP models

- A second test scenario can be obtained with

  `(UC1 [] P(ts1)) [T= UC1T`

  where **`P(t)`** generates a process whose maximum trace is **`t`** itself

- Example: **`P(ts1) = goToMessageCenter -> hotMessageFolderIsDisplayed -> ... -> STOP`**

- The second test scenario is:

```
ts2 = <goToMessageCenter,
  hotMessageFolderIsDisplayed, ...
  messageStorageIsFull, ...>
```

Test generation from CSP models

- In general ...

```
(S [] P(ts1) [] ... P(tsn)) [T= ST
```

- In practice this can be automated by, for instance, iteratively invoking FDR in background
- A prototype implementation: the ATG tool
- Exercise: generate all the test scenarios that correspond to successful termination of UC1I

CSP Test Purpose

- A type of selection directive
- Partial specification of the desired tests
- Deterministic
- Complete (wrt the alphabet of the implementation)
- Includes marking events

  accept, refuse

Test selection with Test Purpose

- Let S be a test model and TP a test purpose. Then

```
ST = S |[AlphaS]| TP
```

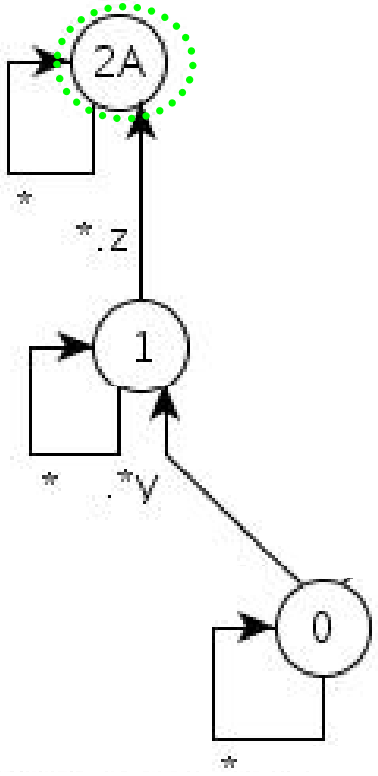
- Test scenarios can then be obtained from ST as before, as counter examples of

```
S [T= ST
```

Example: test selection with Test Purpose

Accept – select target behavior

*y*z



TP = UNTIL(AlfaS, {y},

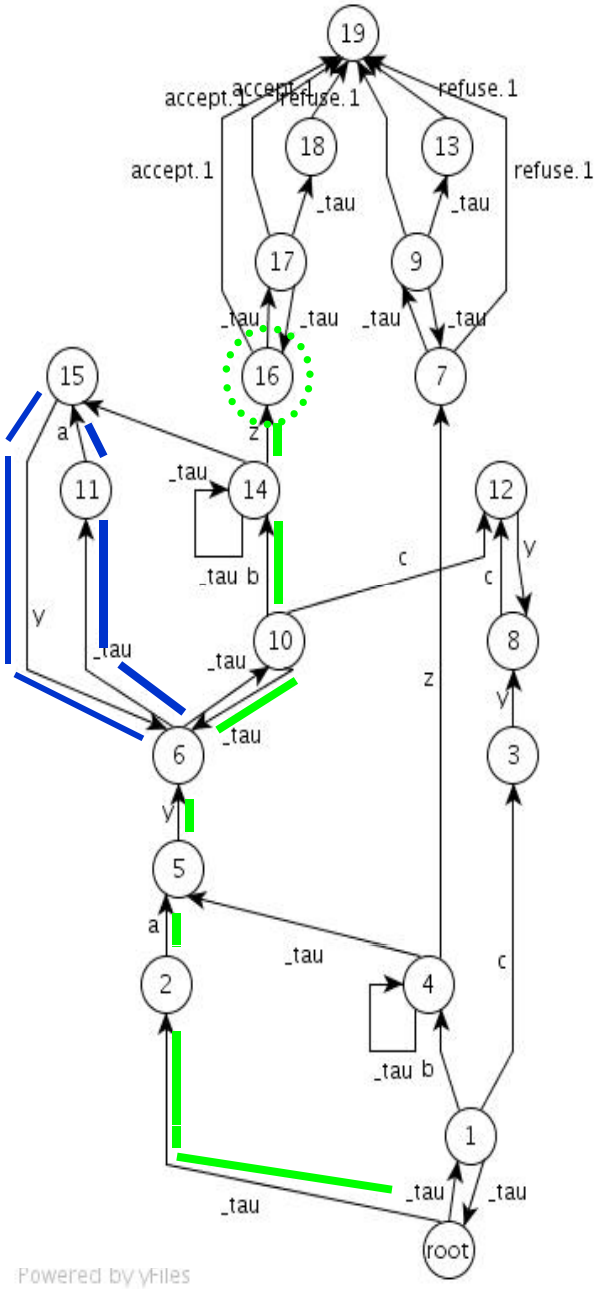
UNTIL(AlfaS,{z},

ACCEPT

)

)

Example: test selection with Test Purpose



PRODUCT = SYSTEM |[AlfaS]| TP

- Generation example

```
SYSTEM [T= PRODUCT

TS1 = <a,y,b,z,accept>

TS2 = <a,y,a,y,b,z,accept>

TS3 = ?

How many test cases?
```

Soundness

- Addressing soundness involves
  - Conformance notion
  - Definition of sound test cases (suites)
  - Generation of sound test cases from test scenarios

CSP Input-Output Conformance

- Proposed implementation relation based on **ioco**

- Informally:
  - *SUT* conforms to *S* if after each trace of *S*, *SUT* exhibits only outputs that are possible in *S*

BUT ... currently we do not consider quiescence

CSP Input-Output Conformance

- *Definition. SUT* **cspioco** *S* iff

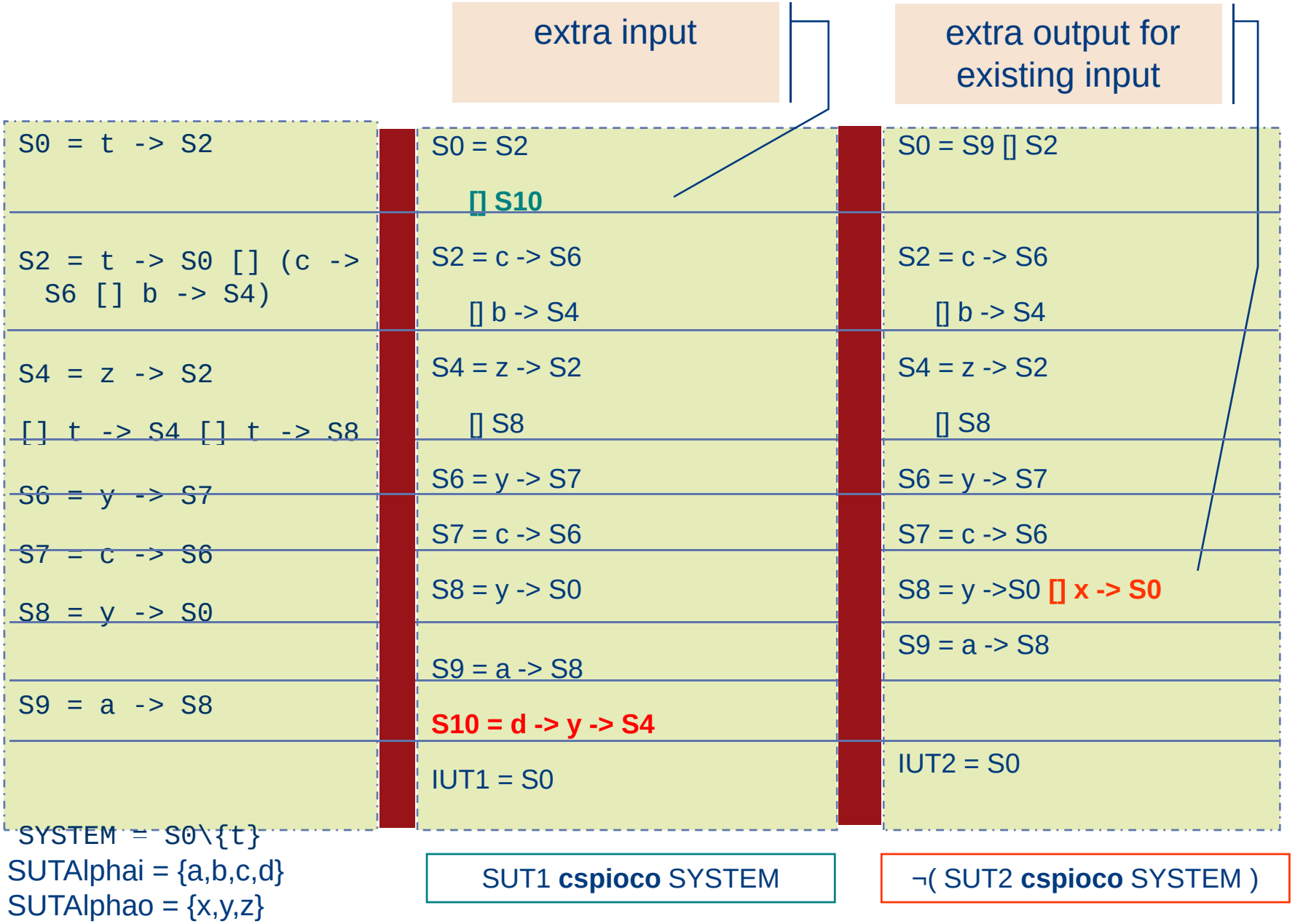
$$\forall\ s \in traces(S) \bullet$$

$$Out(SUT, s) \subseteq Out(S, s)$$

- *Theorem. SUT* **cspioco** S iff

$$S\ [T=\ (S\ |||\ RUN(AlfaIUTo))\ [|AlfaIUT|]\ SUT$$

CSP Input-Output Conformance



Sound Test Case

- Specified as a CSP process, say TC
  - interacts with any implementation that can be modeled as a CSP process (hypothesis)
  - is input complete wrt the SUT alphabet
- Test outputs are implementation inputs (and the other away round)
- Constructed from a trace selected from S (test scenario *ts*)

Sound Test Case

- Sample of a sound CSP test case from test scenario
  <a,y,b,z> Input = {a,b,c,d} Output = {x,y,z}

```
TC  = a -> TC1
TC1 = y -> TC2
      [] e : Output - {y} @ e -> FAIL
TC2 = b -> TC3
TC3 = z -> PASS
      [] y -> INCONCLUSIVE
      [] e : Ouput - {z,y} @ e -> FAIL
```

Verdicts

```
PASS = pass -> STOP

INC = inc -> STOP

FAIL = fail -> STOP
```

Test execution

- Test execution in CSP

$$\mathbf{EXEC = } \mathit{\mathbf{SUT}} \mathbf{\ |[AlphaSUT]|\ TC}$$

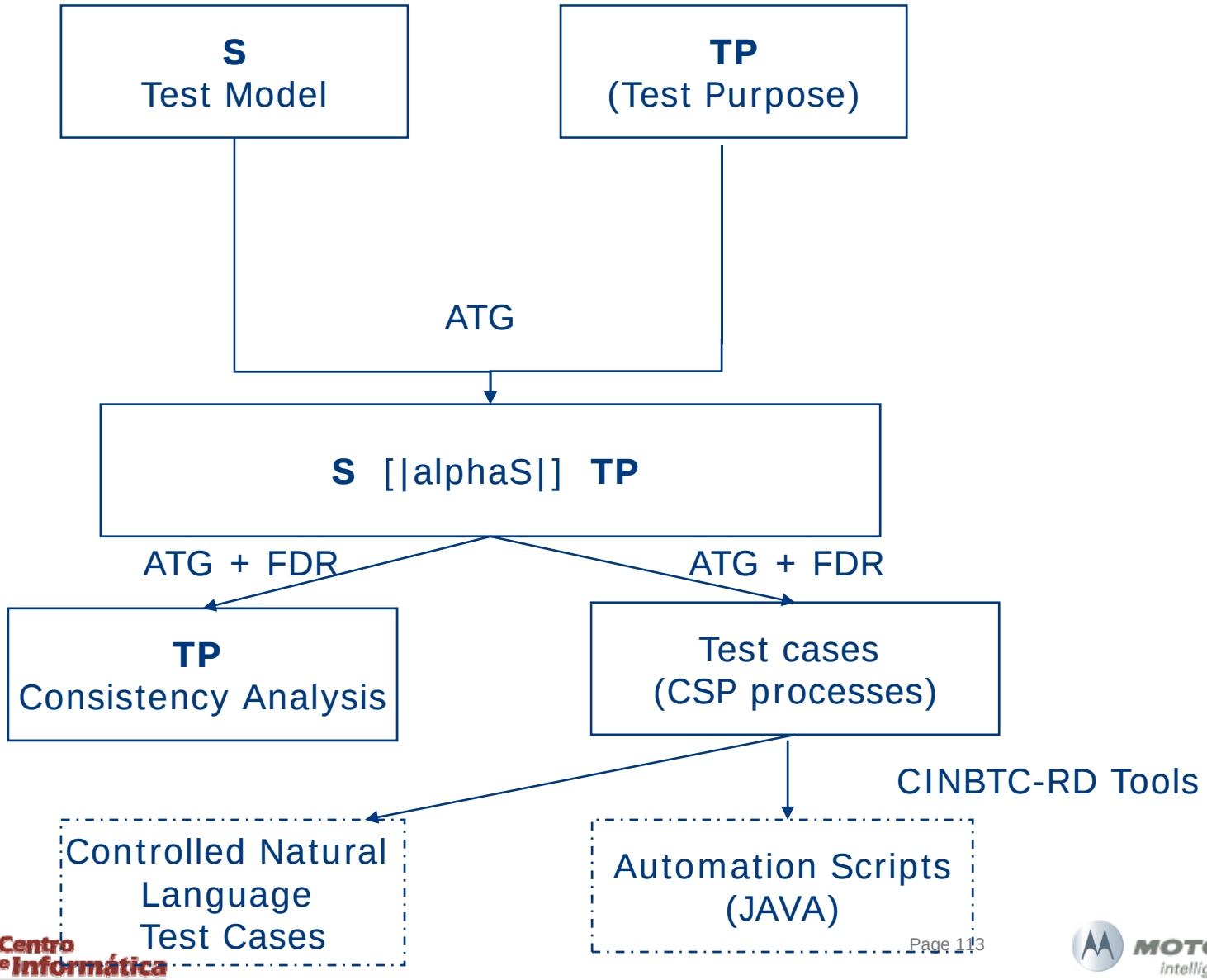
- The execution leads to a fail verdict in at least one trace, if the following holds

$$\mathbf{(EXEC \setminus AlphaSUT)\ [T=\ FAIL}$$

- Soundness in terms of process refinement

$$\neg\exists \mathit{\mathbf{SUT}} \bullet \mathit{\mathbf{SUT}}\ \mathbf{cspioco\ S} \wedge$$
$$\mathbf{(EXEC \setminus AlphaSUT)\ [T=\ FAIL}$$

The ATG tool



Summary

- Process algebraic characterisation of a guided test generation approach
- Formalisation and tool support
- Advantages
  - Uniformity: a single formalism
  - Modularity: model, TP and TC (de)composition
  - Generation via model (refinement) checking

    In principle, productivity …
- Disadvantages
  - State space explosion
  - No syntactic control over the generation process
  - Portability (FDR-Unix/Linux)

Related Work

- Tretmans (IOCO)
- Jard and Jéron (TGV)
- Peleska and Siegel
- Schneider
- Cavalcanti and Gaudel

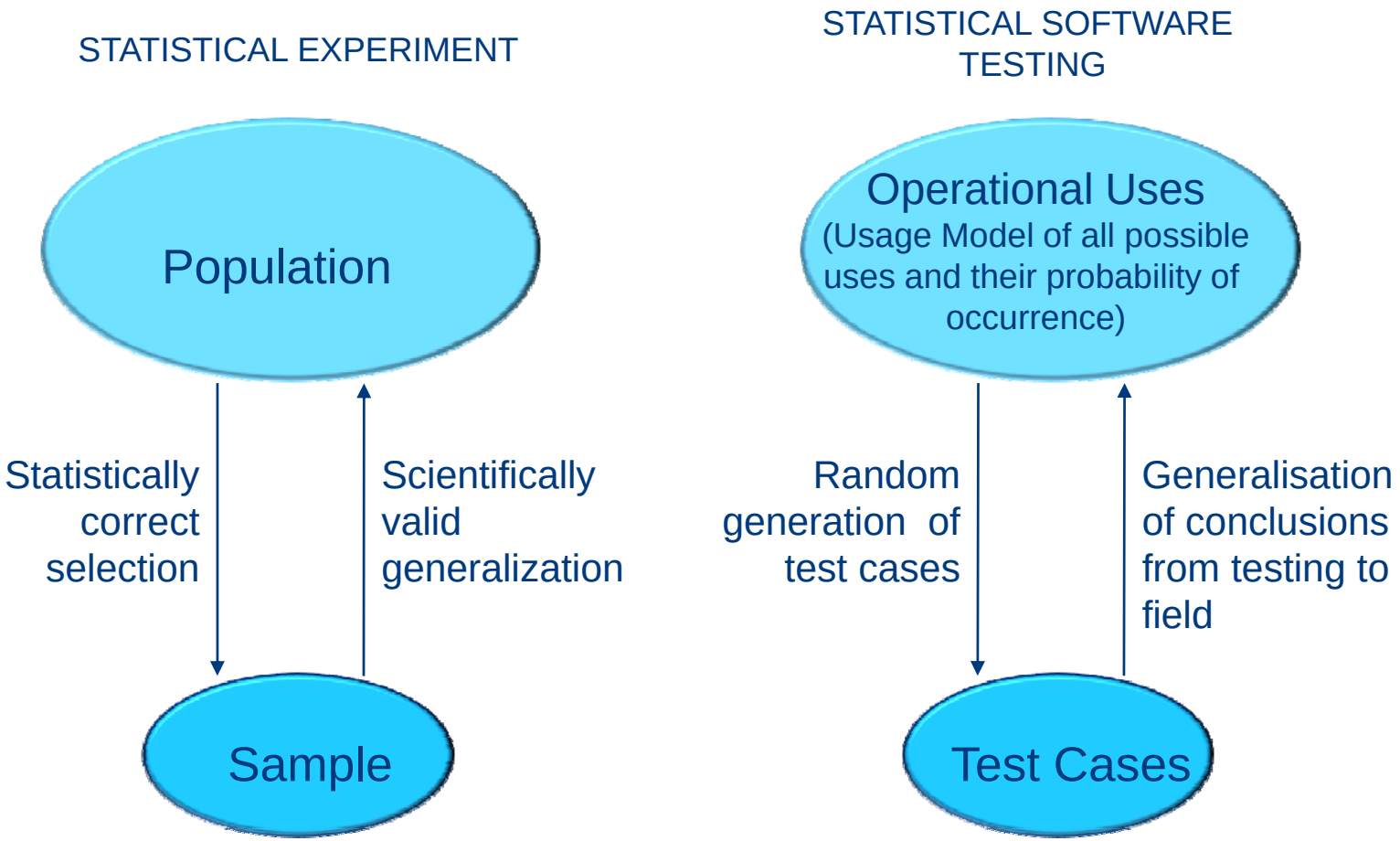
Future work

- Data abstraction to deal with state space explosion
- Explore more elaborate CSP semantic models and consider quiescence
- Selection approaches for component testing based on system architecture

TEST MODELS AND TEST GENERATION

- Input-Output Labelled Transition Systems
- Annotated Labelled Transition Systems
- Process Algebra
- Markov Chains

Statistical Testing



Combinatorial Explosion in Possible Usage Scenarios

Assumptions

- A usage scenario is a sequence of 1 to 10 inputs
- 20 different inputs are possible
- Inputs can be repeated

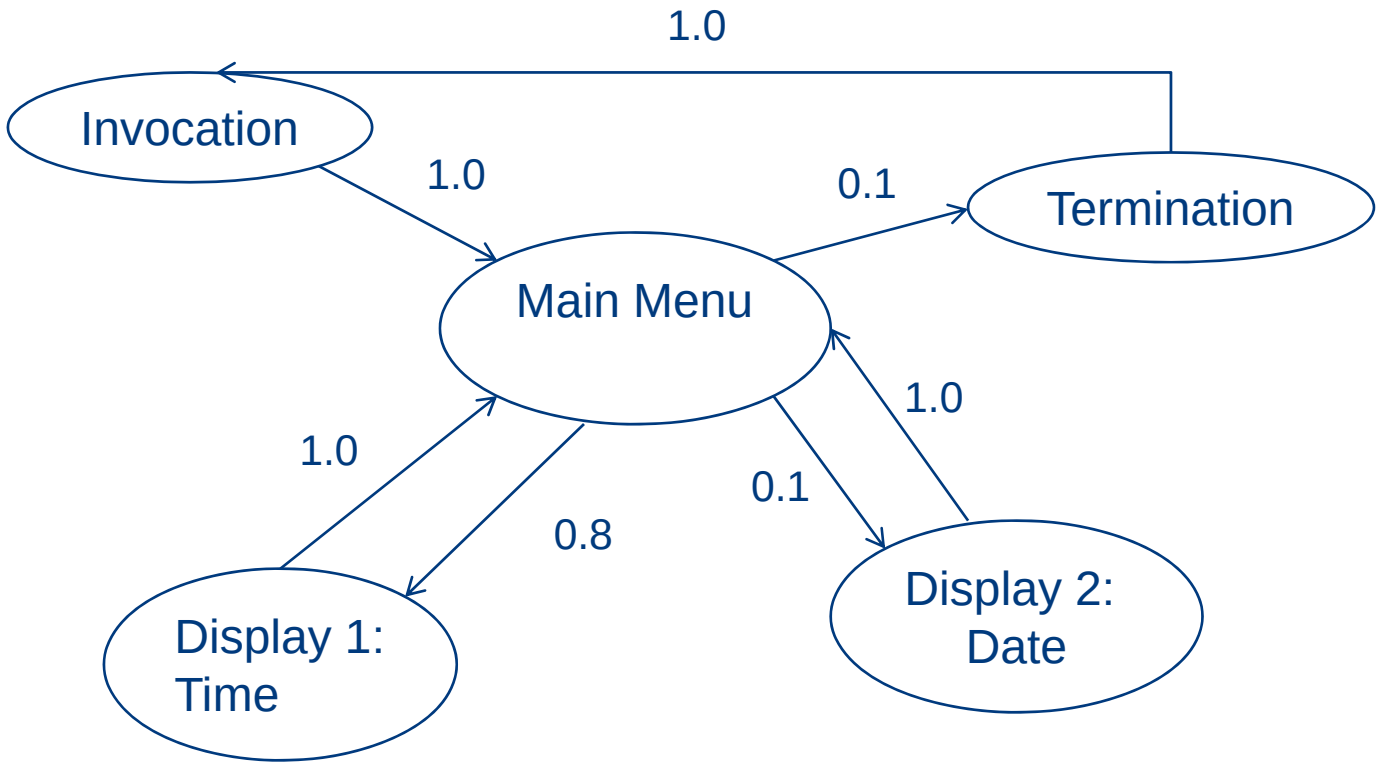
Length of Input Sequence	Number of Possible Usage Scenarios	
1	20	= 20
2	20 x 20	= 400
3	20 x 20 x 20	= 8000
4	20 x 20 x 20 x 20	= 160,000
5	20 x 20 x 20 x 20 x 20	= 3,200,000
6	20 x 20 x 20 x 20 x 20 x 20	= 64,000,000
7	20 x 20 x 20 x 20 x 20 x 20 x 20	= 1,280,000,000
8	20 x 20 x 20 x 20 x 20 x 20 x 20 x 20	= 25,600,000,000
9	20 x 20 x 20 x 20 x 20 x 20 x 20 x 20 x 20	= 512,000,,000,000
10	20 x 20 x 20 x 20 x 20 x 20 x 20 x 20 x 20 x 20	= 10,240,000,000,000
Total Usage Scenarios		= 10,778,947,368,420

Figure from [Tra95]

The Model

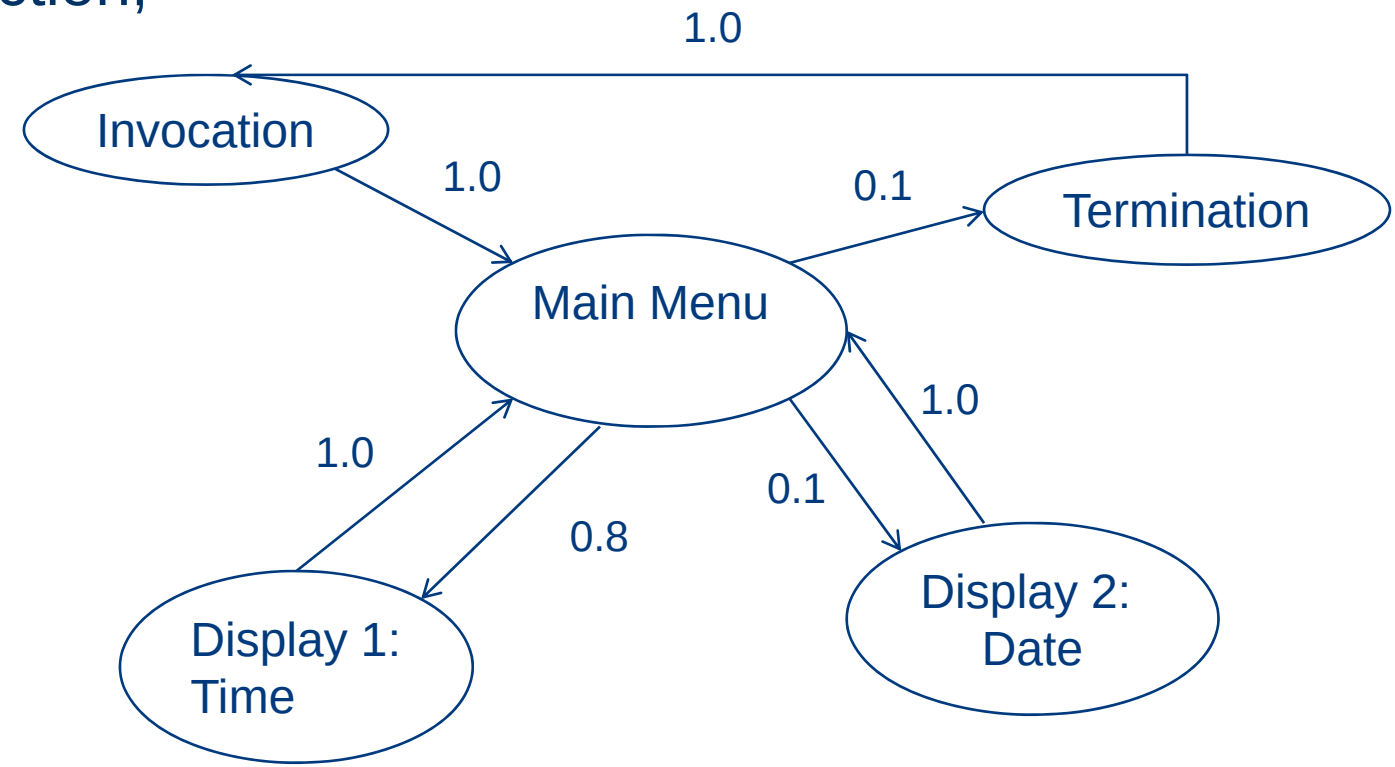
- A usage chain consists of:
  - Nodes that represent usage states, i.e., the system visible modes of operation;
  - Arcs that represent transitions between states;
  - Probability distribution to transitions that usually indicates the best estimate of real usage
  - Two special states are added: a start and a termination state
- The process of constructing this chain usually involves three steps:
  1. Identify and represent states and transitions in a diagram;
  2. Assign probabilities to the transitions;
  3. Validate the model by transition coverage analysis.

A Usage Model



Test Case Generation

- Test cases are selected by traversing the chain from the start to the termination node;
- The next transition is chosen by a probability distribution function;



Benefits from Statistics

- Statistical methods can be applied to help test planning and reliability assessment by providing measurements such as:
  - Probability of a transition appearing in a test sequence;
  - Expected number of sequences to cover a state or transition;
  - Expectation and variance of state or transition first passage;
  - Number of sequences needed to cover all states and transitions;
  - Number of statistically typical usage paths through the software;
  - And so on.

Benefits from Statistics

- This information can help to define:
  - Optimal test allocation;
  - Estimate variance of overall failure rate;
  - Probability of finding a failure;
  - Expected number of failures;
  - And so on.

Assumptions that Underlie the Validity of Inferences

- Each trial is performed under the same conditions
  - Software, Input Values, Environment, Basis for Evaluation, Tester.
- There is one outcome per trial
  - Success or Failure
- All outcomes are possible in each trial
  - All possible scenarios are candidate for selection
  - Testing should not proceed in the presence of blocking failures
- Trials are independent
  - The inputs and outputs of a test case have no bearing on the inputs and outcomes of any subsequent test case.

Final Remarks

- Statistical testing can be a very valuable tool for testers;
- Markov chains and probability distributions are difficult to construct;
- Automatic generation of usage chains and probability distribution from UML sequence diagrams and usage profiles (SPACES tool).
- An even distribution can be a start point and can be still more effective than deterministic choice.

TEST MODEL GENERATION FROM USE CASE SPECIFICATIONS

- Use Case Templates
- Generating ALTS Models

Centro de Informática U·F·P·E

MOTOROLA intelligence everywhere™
Mobile Devices Software Group

Use Case Specifications as Test Models

- Test designers tend to adopt informal specifications
- Potential ambiguities and inconsistencies
- Formal models are not usually an acceptable solution
- Use case specifications as an alternative
  - templates (“control flow”)
  - *Controlled Natural Language* – CNL (text processing)

Use Case Specifications as Test Models

- Several levels of abstraction
  - User view
  - Component View, …
- Together with the CNL, the templates allow automatic test model generation
  - ALTS
  - CSP, …
- Focus on User View and ALTS generation

TEST MODEL GENERATION FROM USE CASE SPECIFICATIONS

- Use Case Templates
- Generating ALTS Models

Use Case Template

Feature 11111 – My Phonebook

id Name

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.		My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Use Cases

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Use Case Template

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Use case Id and Name

Description

This use case describes the creation of a new contact.

A brief description

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.		My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Use Case Template

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.		My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Main Flow

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Exception Flow

Use Case Template

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

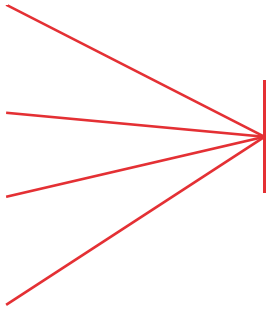
Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.		My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.



A sequence of steps

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Use Case Template

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

A brief description

Main Flow

Description: Create a new contact
From Step: START
To Step: END

The list of *From* and *To* steps

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.		My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Related requirements

MOTOROLA
intelligence everywhere™
Mobile Devices Software Group

TEST MODEL GENERATION FROM USE CASE SPECIFICATIONS

- Use Case Templates
- Generating ALTS Models

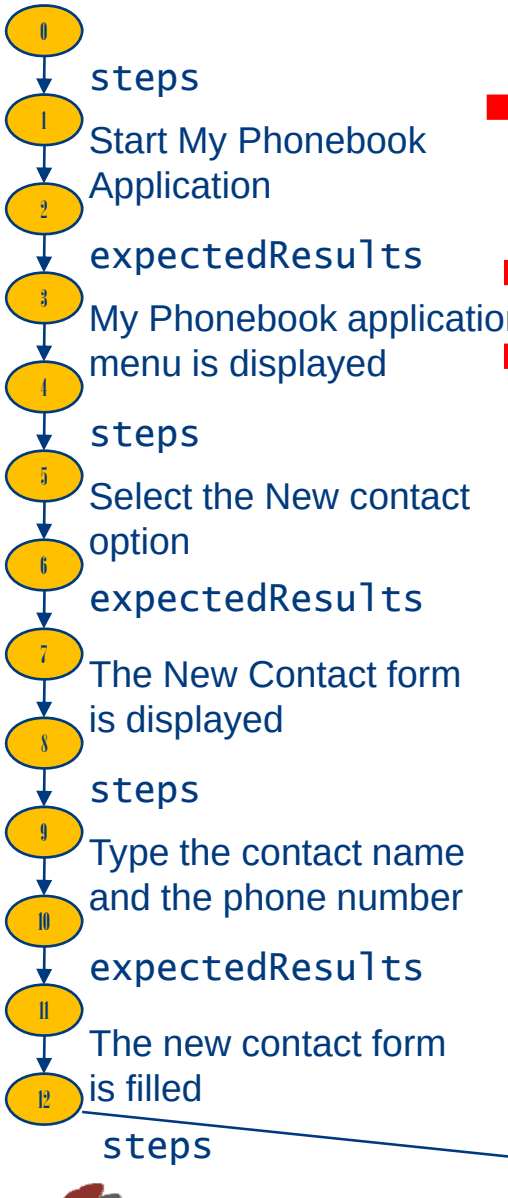
Overview of the Algorithm

- Use case flows processed sequentially
  - Main flow is the starting point
  - Each step gives rise to states and transitions
  - Order of steps is preserved
- The current state is:
  - the last state created in case the ***From Step*** field is defined as **START** or this is the first state; or
  - the last state of a given step (defined in the ***From Step*** field) of another template
- ***From step*** and ***To Step*** guide the connection of each trace created by each of the templates.

Overview of the Algorithm

- ***User Action***, ***System State*** and ***System Response*** become transitions
  - preceded by ***steps***, ***conditions*** and ***expectedResults*** annotations, respectively
- States are created as new transitions need to be added
- States already created can be reused when:
  - ***To Step*** is different from **END**;
  - ***From Step*** is different from **START**;
  - A new condition is considered based on a user action already added
- Duplicated annotated transitions are also avoided

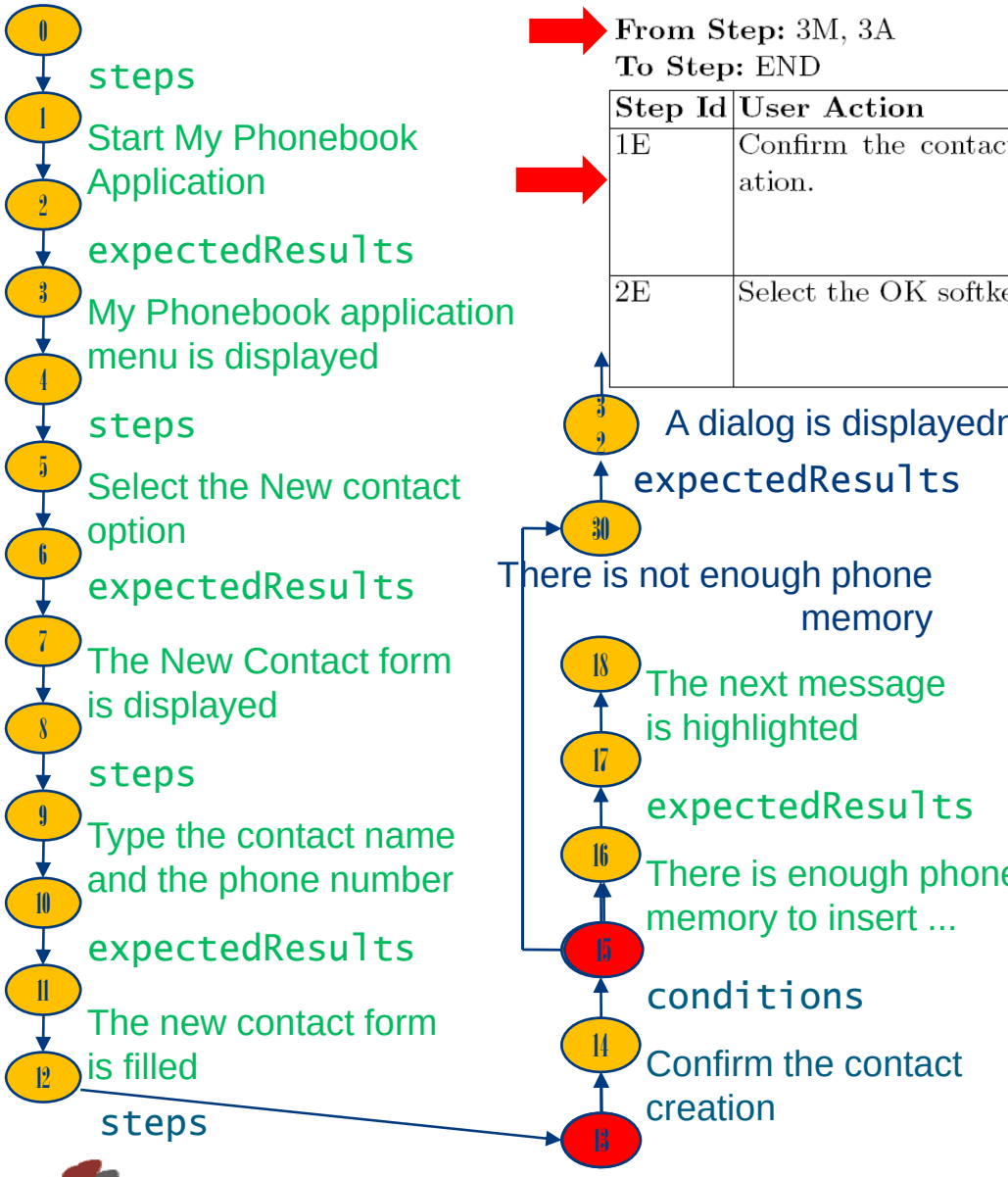
Generating ALTS Models



Main Flow
Description: Create a new contact **From Step:** START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.		My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	The next message is highlighted.

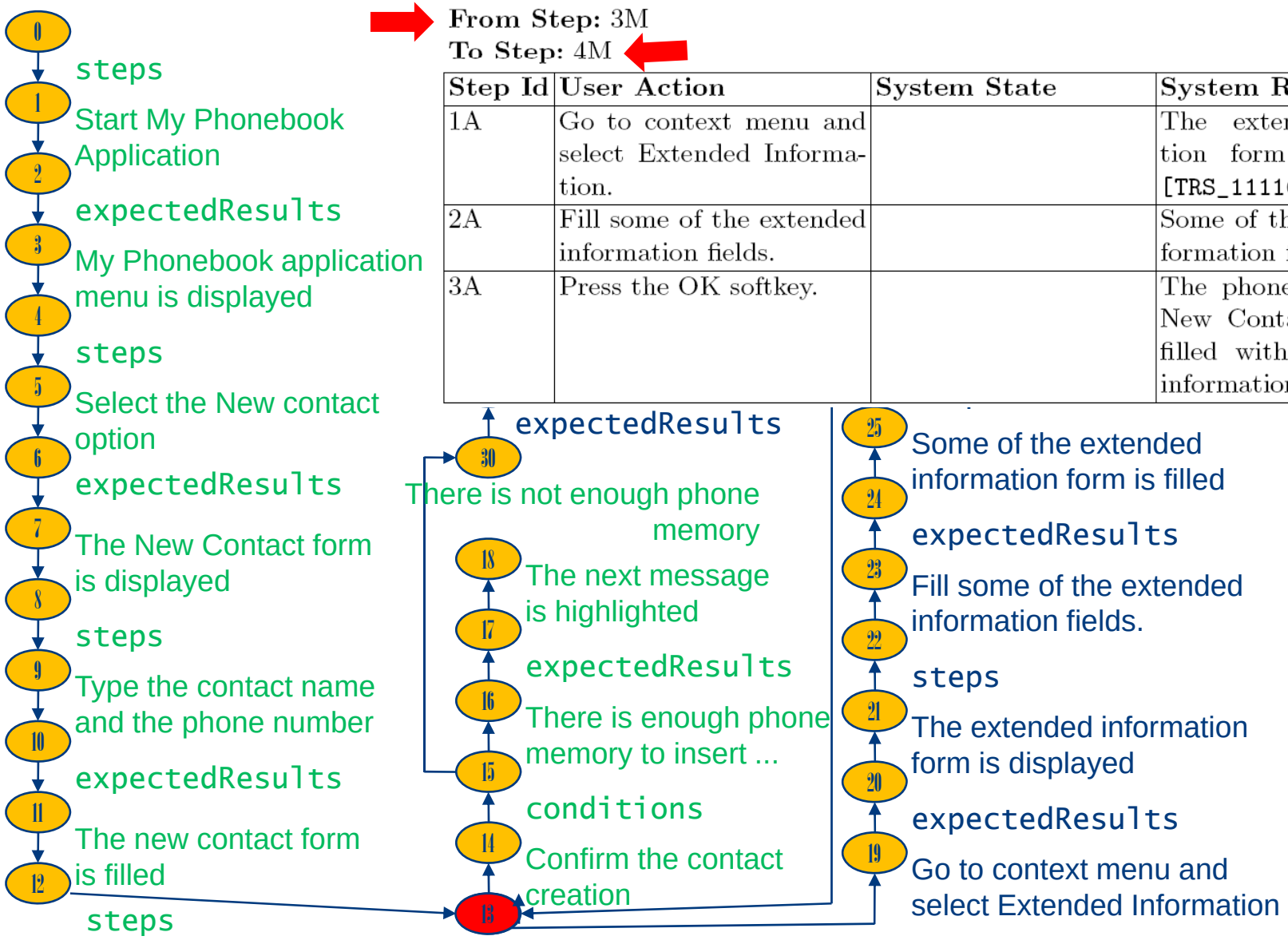
Generating ALTS Models



From Step: 3M, 3A
To Step: END

Step Id	User Action	System State	System Response
1E	Confirm the contact creation.	There is not enough phone memory.	A dialog is displayed informing that there is not enough memory. [TRS_111166_103]
2E	Select the OK softkey.		The phone goes back to My Phonebook application menu.

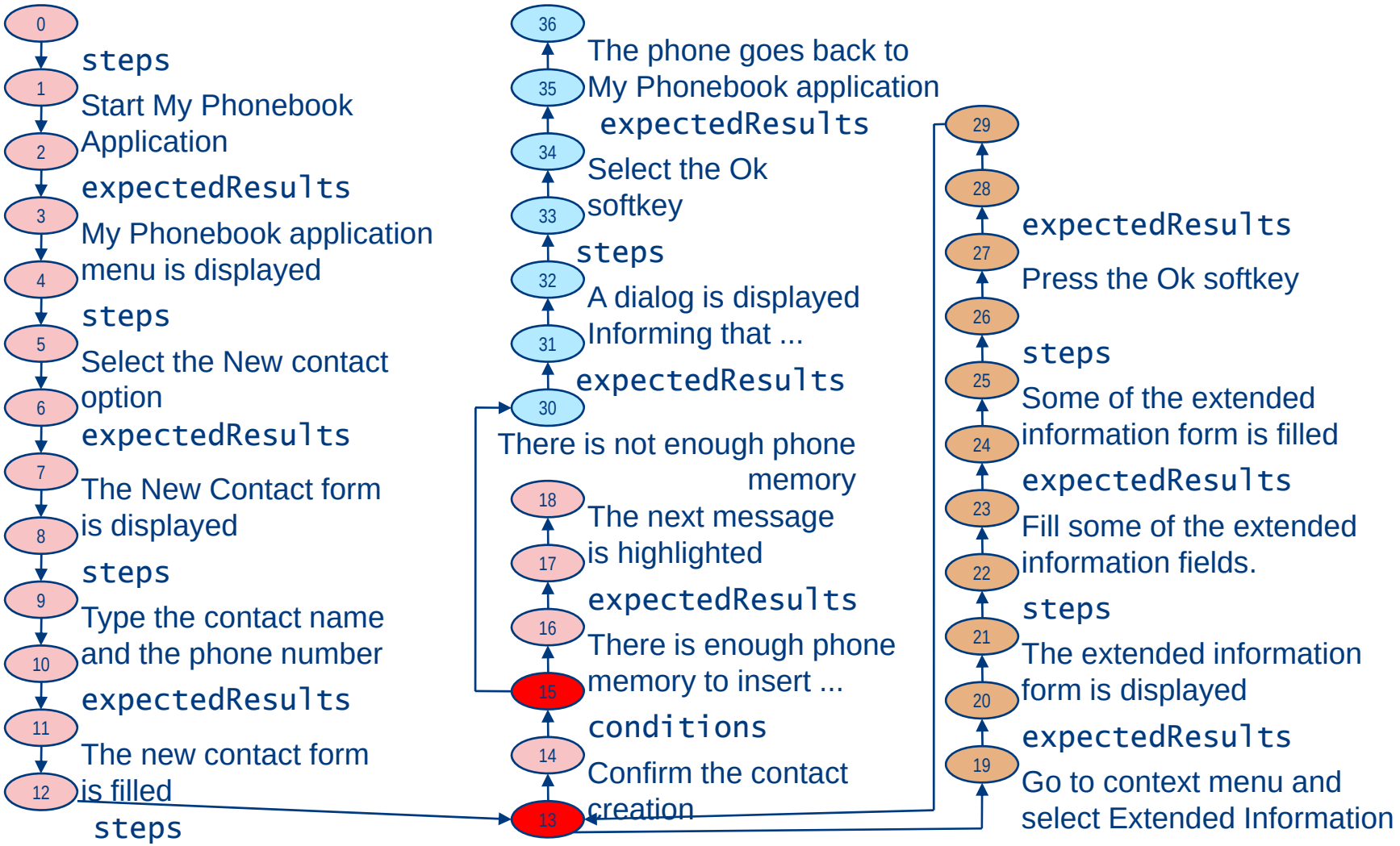
Generating ALTS Models



From Step: 3M
To Step: 4M

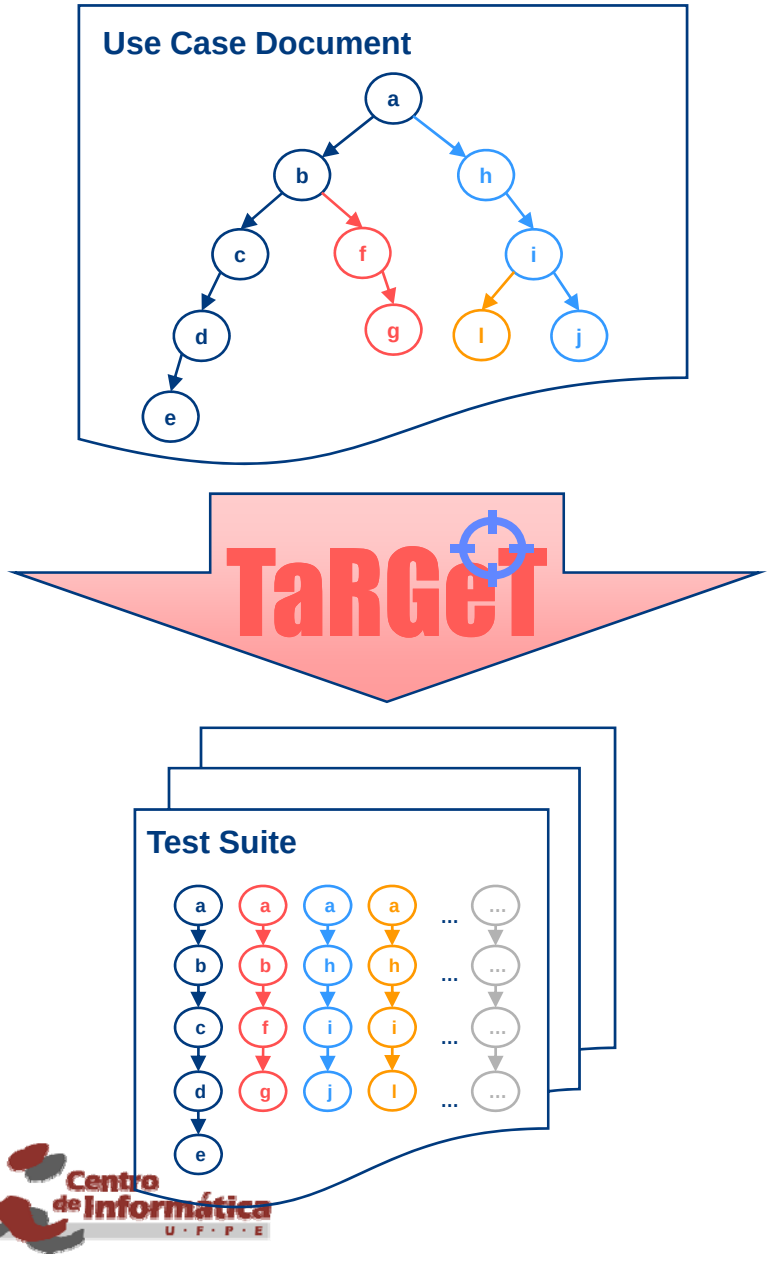
Step Id	User Action	System State	System Response
1A	Go to context menu and select Extended Information.		The extended information form is displayed. [TRS_111166_102]
2A	Fill some of the extended information fields.		Some of the extended information form is filled.
3A	Press the OK softkey.		The phone goes back to New Contact form. It is filled with the extended information.

Generating ALTS Models



THE TARGET TOOL

TaRGeT Test Generation



- Input
  - Use case documents
- Output
  - Test cases according a template (manual testing)
  - Traceability matrices
- Generation algorithm
  - All scenarios
  - Test selection
    - Requirements
    - Test purposes
    - Similarity

TaRGeT Test Generation

Case	Reg. Level	Exe. Type	Case Description	Procedure	Expected Results
1	na	Man	TC_1	Create a new contact. There is no enough memory.	
			Requirements:	TRS_111166_103	
			Setup:		
			Initial Conditions:	My Phonebook application is installed in the phone.There is no enough phone memory.	
			Notes:	Test case auto-generated by TaRGeT system.	
			Test Procedure:		
			1	Start My Phonebook application.	My Phonebook application menu is displayed.
			2	Select the New Contact option.	The New Contact form is displayed.
			3	Type the contact name and the phone number.	The new contact form is filled.
			4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.
			Final Conditions:		
			Cleanup:		

TaRGeT Test Generation

Case	Reg. Level	Exe. Type	Case Description	Procedure	Expected Results
1	na	Man	TC_1	Create a new contact. There is no enough memory.	
			Requirements:	TRS_111166_103	
			Setup:		
			Initial Conditions:	My Phonebook application is installed in the phone.There is no enough phone memory.	
			Notes:	Test case auto-generated by TaRGeT system.	
			Test Procedure:		
			1	Start My Phonebook application.	My Phonebook application menu is displayed.
			2	Select the New Contact option.	The New Contact form is displayed.
			3	Type the contact name and the phone number.	The new contact form is filled.
			4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.
			Final Conditions:		
			Cleanup:		

Test case template

Case	Reg. Level	Exe. Type	Case Description	Procedure	Expected Results
1	na	Man	TC_1	Create a new contact. There is no enough memory.	
			Requirements:	TRS_111166_103	
			Setup:		
			Initial Conditions:	My Phonebook application is installed in the phone.There is no enough phone memory.	
			Notes:	Test case auto-generated by TaRGeT system.	
			Test Procedure:		
			1	Start My Phonebook application.	My Phonebook application menu is displayed.
			2	Select the New Contact option.	The New Contact form is displayed.
			3	Type the contact name and the phone number.	The new contact form is filled.
			4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.
			Final Conditions:		
			Cleanup:		

TaRGeT Test Generation

Case	Reg. Level	Exe. Type	Case Description	Procedure	Expected Results
1	na	Man	TC_1	Create a new contact. There is no enough memory.	
			Requirements:	TRS_111166_103	
			Setup:		
			Initial Conditions:	My Phonebook application is installed in the phone.There is no enough phone memory.	
			Notes:	Test case auto-generated by TaRGeT system.	
			Test Procedure:		
			1	Start My Phonebook application.	My Phonebook application menu is displayed.
			2	Select the New Contact option.	The New Contact form is displayed.
			3	Type the contact name and the phone number.	The new contact form is filled.
			4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.
			Final Conditions:		
			Cleanup:		

TaRGeT Test Generation

Case	Reg. Level	Exe. Type	Case Description	Procedure	Expected Results
1	na	Man	TC_1	Create a new contact. There is no enough memory.	
			Requirements:	TRS_111166_103	
			Setup:		
			Initial Conditions:	My Phonebook application is installed in the phone.There is no enough phone memory.	
			Notes:	Test case auto-generated by TaRGeT system.	
			Test Procedure:		
			1	Start My Phonebook application.	My Phonebook application menu is displayed.
			2	Select the New Contact option.	The New Contact form is displayed.
			3	Type the contact name and the phone number.	The new contact form is filled.
			4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.
			Final Conditions:		
			Cleanup:		

TaRGeT Test Generation

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Case Description	Procedure	Expected Results
TC_1		
Requirements:		
Initial Conditions:		
Test Procedure:		

TaRGeT Test Generation

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Case Description	Procedure	Expected Results
TC_1		
Requirements:		
Initial Conditions:		
Test Procedure:		

TaRGeT Test Generation

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Case Description	Procedure	Expected Results
TC_1		
Requirements:		
Initial Conditions:		
Test Procedure:		
1	Start My Phonebook application.	My Phonebook application menu is displayed.
2	Select the New Contact option.	The New Contact form is displayed.
3	Type the contact name and the phone number.	The new contact form is filled.
4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.

TaRGeT Test Generation

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Case Description	Procedure	Expected Results
TC_1		
Requirements:	TRS_111166_103	
Initial Conditions:		
Test Procedure:		
1	Start My Phonebook application.	My Phonebook application menu is displayed.
2	Select the New Contact option.	The New Contact form is displayed.
3	Type the contact name and the phone number.	The new contact form is filled.
4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.

TaRGeT Test Generation

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

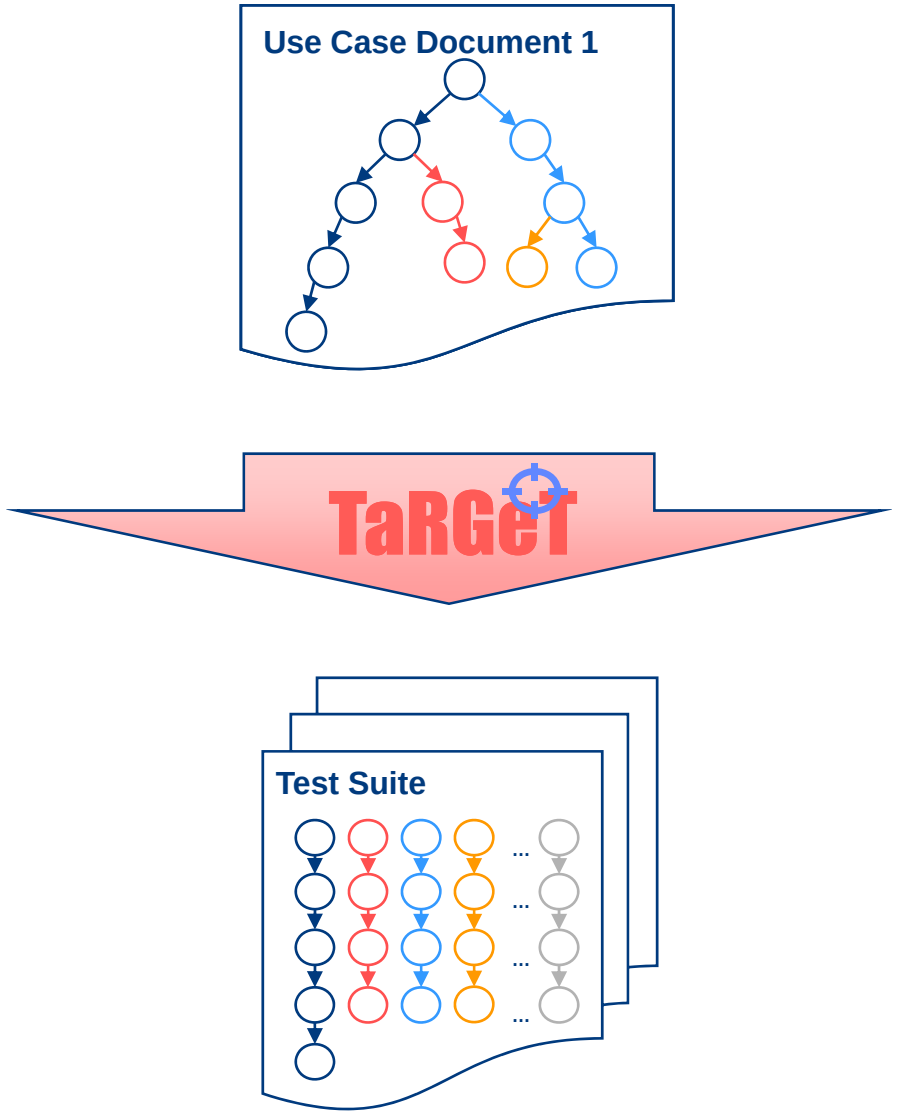
Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

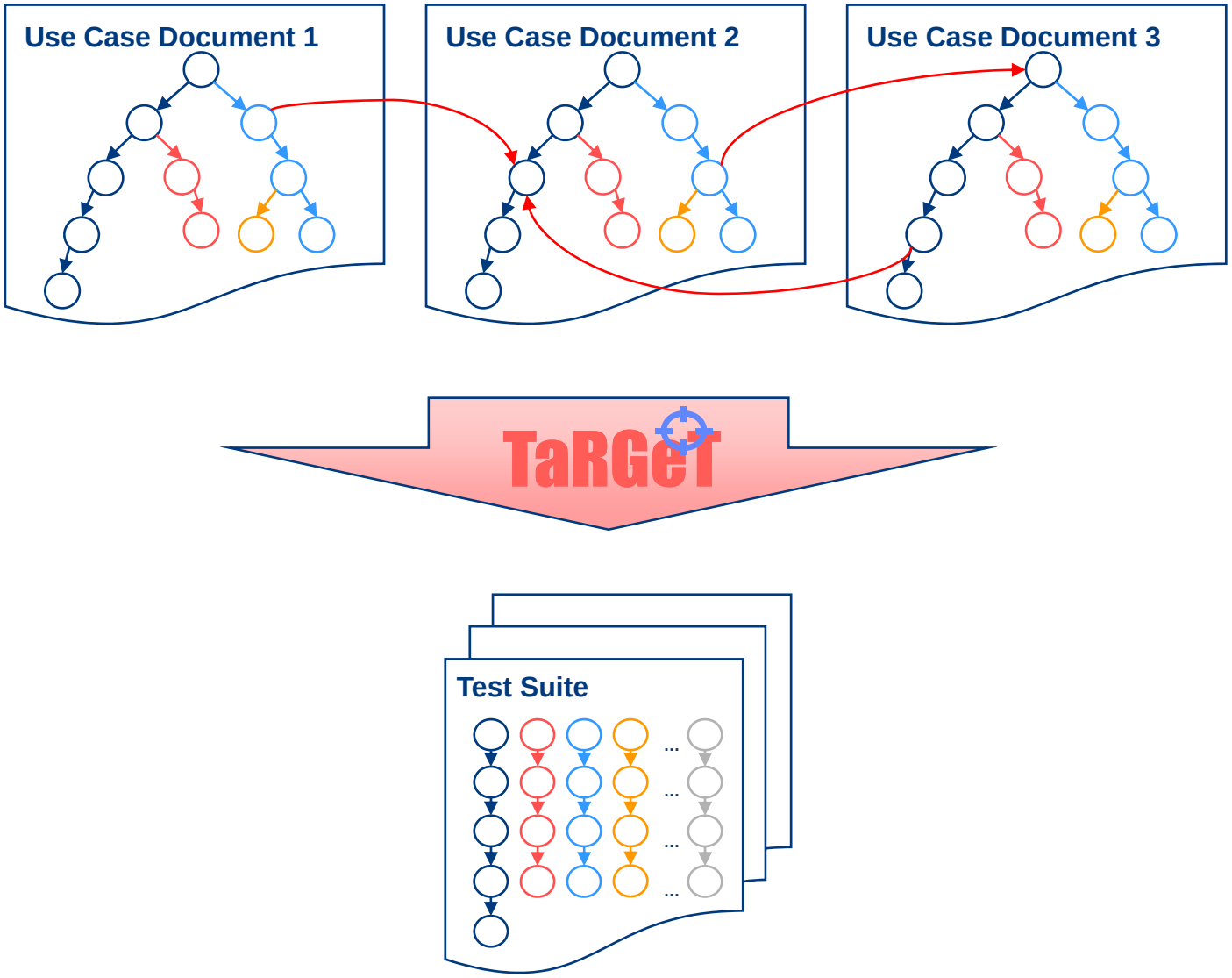
Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Case Description	Procedure	Expected Results
TC_1		
Requirements:	TRS_111166_103	
Initial Conditions:	My Phonebook application is installed in the phone.There is no enough phone memory.	
Test Procedure:		
1	Start My Phonebook application.	My Phonebook application menu is displayed.
2	Select the New Contact option.	The New Contact form is displayed.
3	Type the contact name and the phone number.	The new contact form is filled.
4	Confirm the contact creation.	A dialog is displayed informing that there is no enough memory.

TaRGeT Test Generation



TaRGeT Test Generation



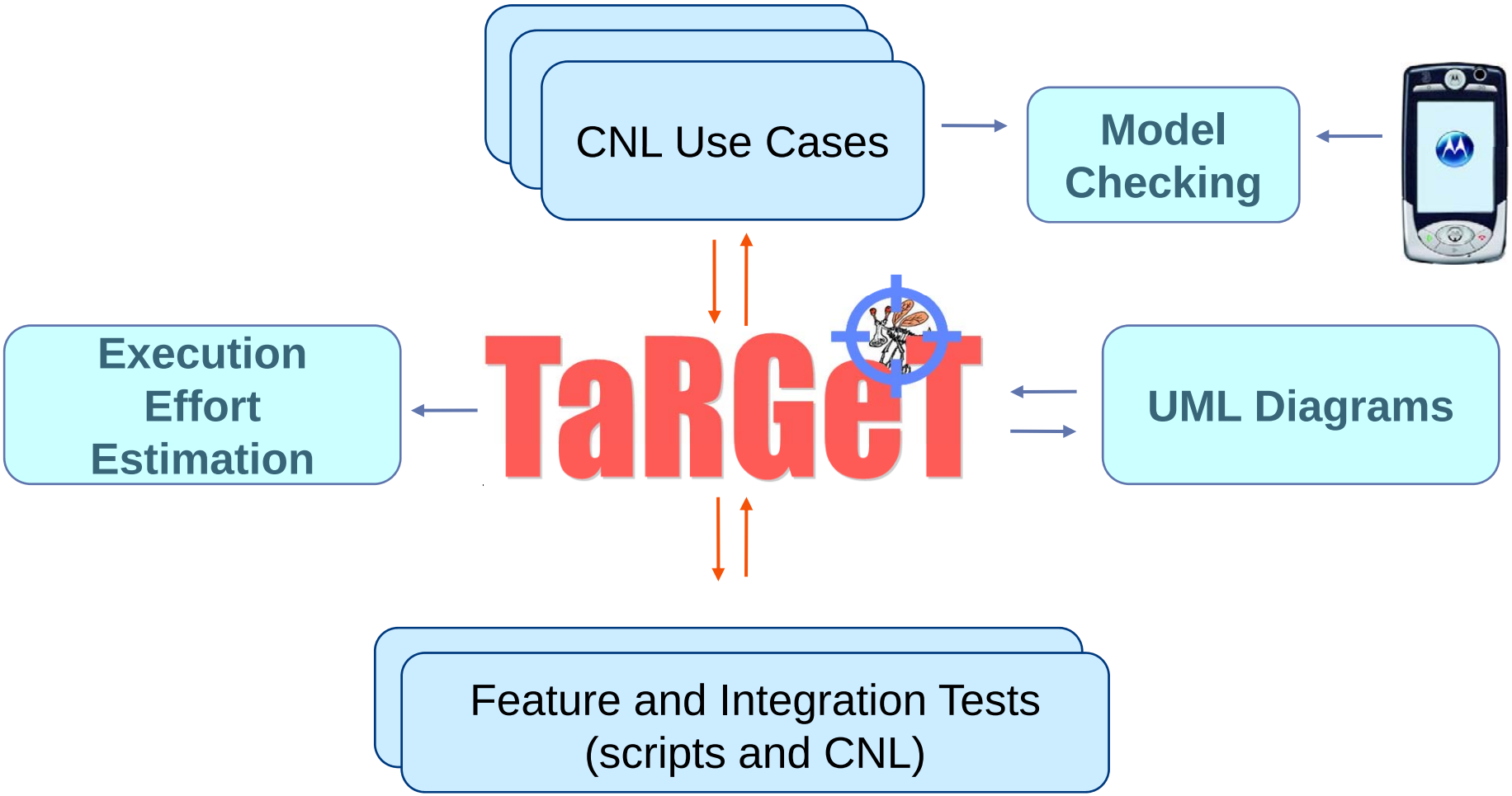
TaRGeT Demonstration

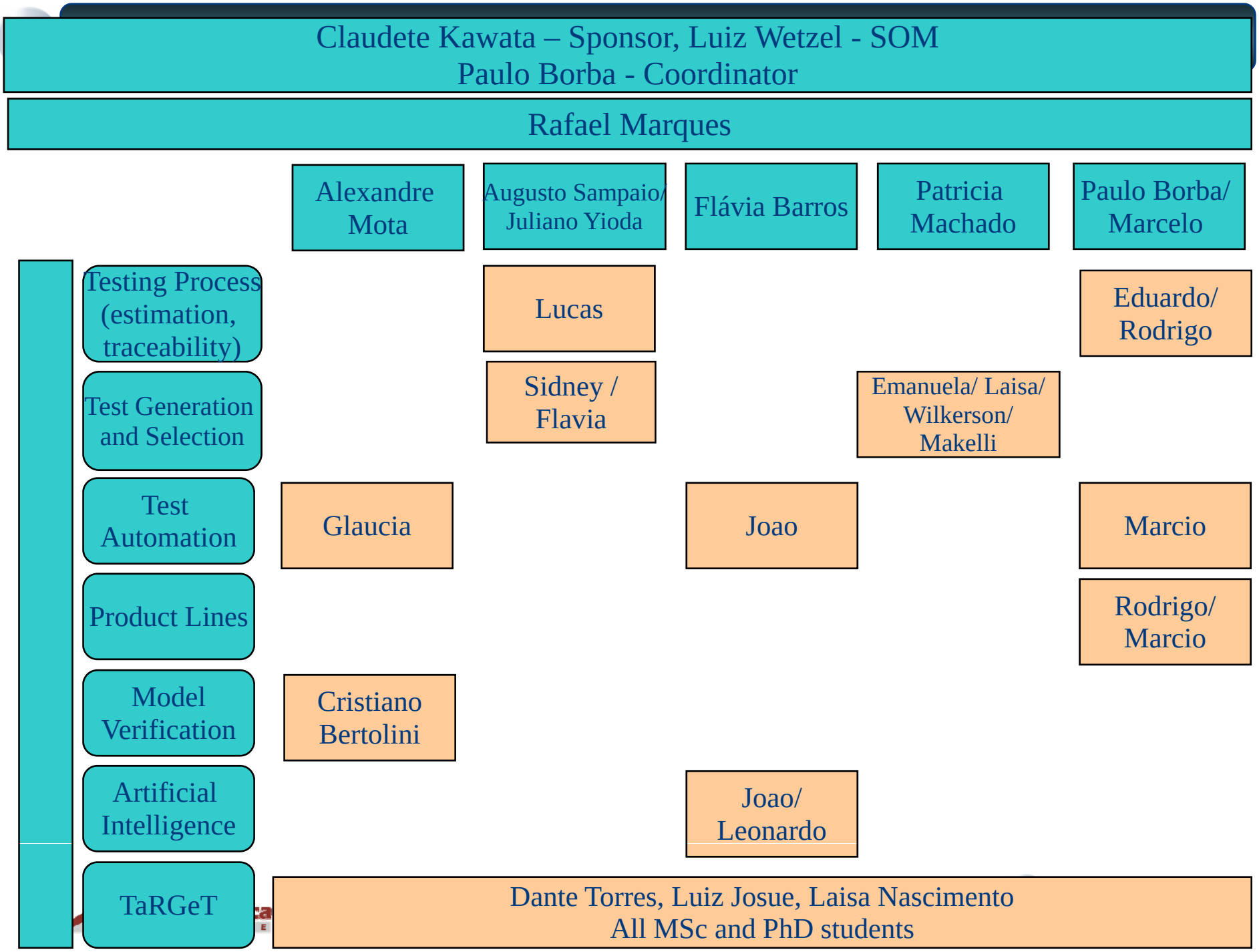
CONCLUDING REMARKS

Summary

- Focus on approaches to MBT
  - test models
  - test generation
  - test selection
- Our solution
  - hidden formal methods
  - Experiments
    - For small features: around 50% increase in productivity
    - More expected for larger features

Brazil Test Center Research Group




Claudete Kawata – Sponsor, Luiz Wetzel - SOM
Paulo Borba - Coordinator
Rafael Marques
Alexandre Mota
Augusto Sampaio/ Juliano Yioda
Flávia Barros
Patricia Machado
Paulo Borba/ Marcelo
Testing Process (estimation, traceability)
Lucas
Eduardo/ Rodrigo
Test Generation and Selection
Sidney / Flavia
Emanuela/ Laisa/ Wilkerson/ Makelli
Test Automation
Glaucia
Joao
Marcio
Product Lines
Rodrigo/ Marcio
Model Verification
Cristiano Bertolini
Artificial Intelligence
Joao/ Leonardo
TaRGeT
Dante Torres, Luiz Josue, Laisa Nascimento
All MSc and PhD students

REFERENCES

References

- [AM07] Andrade, W.L., Neto, F.G.O., Machado, P.D.L.: Geração de casos de teste de interrupção para aplicações de celulares. In: Proceedings of the VIII Workshop de Teste e Tolerância a Falhas (WTF 2007). (2007) 129–142.
- [CS06] Cabral, G., Sampaio, A.: Formal specification generation from requirement documents. In: Brazilian Symposium on Formal Methods (SBMF). (2006) 217–232.
- [COM07] Cartaxo, E.G., de Oliveira Neto, F.G., Machado, P.D.L.: Automated test case selection based on a similarity function. In: Proceedings of MOTES07 – Model based Testing - Workshop in conjunction with the 37th Congress of the Gesellschaft fuer Informatik. Volume 110 of LNI (2007) 381–386.
- [Car08+] Cartaxo, E.G., Andrade, W.L., Neto, F.G.O., Machado, P.D.L.: Lts-bt: A tool to generate and select functional test cases for embedded systems. In: To be published in Proceedings of the ACM/SAC 2008. (2008).
- [FAM06] de Figueiredo, A.L.L., Andrade, W.L., Machado, P.D.L.: Generating interaction test cases for mobile phone systems from use case specifications. In: A-MOST ’06: Proceedings of the Second Int. Work. on Advances in Model-based Software Testing, New York, NY, USA, ACM Press (2006).

References

- [JJ05] Jard, C., Jéron, T.: Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. Int. J. Softw. Tools Technol. Transf. 7(4) (2005) 297–315
- [Nog07+] Nogueira, S., Cartaxo, E.G., Torres, D.G., Aranha, E.H.S., Marques, R.: Model based test generation: An industrial experience. In: Proceedings of 1st Brazilian Workshop on Systematic and Automated Software Testing, Sociedade Brasileira de Computação (SBC) (2007)
- [Nog06] Nogueira, S.: Geração automática de casos de teste CSP dirigida por propósitos. Master's thesis, Universidade Federal de Pernambuco (UFPE) (2006)
- [Tra95] Trammell, C.: Quantifying the Reliability of Software: Statistical Testing Based on Usage Model. 2nd IEEE Software Engineering Standards Symposium, 1995.
- [Tre96] Tretmans, G.J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools 3 (1996) 103–120