AdaCore

2016-05-11

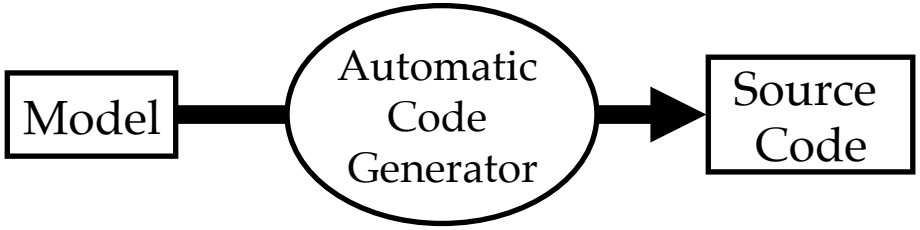
TQL-1 Qualification of a Model-Based Code Generator

S. Tucker Taft
Director of Language Research

HCSS 2016, Annapolis, MD

Model-driven Development of Critical Software

- An Automatic Code Generator (ACG) produces the source code from a model-based specification



- To avoid verifying the generated code the ACG must be "Qualified" (i.e. Trusted) (or “Qualifiable”)
- **Qualification requires a rigorous and thorough verification of the ACG**
  - **Extensive testing of the ACG with high exhaustiveness**

Testing is a major cost in ACG qualification

DO-178C, Tool Qualification Levels

Table 12-1 Tool Qualification Level Determination

Software Level	Criteria		
	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

Failure Condition is …

Level A: Catastrophic

Level B: Hazardous

Level C: Major

Level D: Minor

Criteria 1: A tool whose output is part of the airborne software and thus could insert an error.

Criteria 2: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:

- *Verification process(es) other than that automated by the tool, or*
- *Development process(es) that could have an impact on the airborne software.*

Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error.

Advantages of TQL-1 tool with respect to TQL-4/5 verification tools

- **TQL-1 is about *correctness by construction***

- **TQL-4/5 implies *a posteriori checks***
  - DO-178C/331 Table A-5 (verification of the source code)
    - After the final model is verified it can find source code is not traceable/compliant
  - DO-178C/331 Table A-6 (testing)
    - After the final model is verified it can find executable is not robust/compliant
  - DO-178C/331 Table A-7 (coverage)
    - After the final model is verified it can find uncovered source code

- **A Posteriori verifications may require changes *late in the project***
  - User may need to modify (after source code generation)
    - The generated source code
    - The model
    - The tests
    - The requirements
  - Forced to change them not because of deficient design, but because of problems with the code generation tools

Sidebar: What are DO-178C, DO-330, DO-331, DO-332, DO-333?

- **DO-178C Software Considerations in Airborne Systems**
  - primary document by which the certification authorities such as FAA, EASA and Transport Canada approve all commercial software-based aerospace systems.

Supplements to DO-178C:

- **DO-330 Software Tool Qualification Considerations**
  - addressing qualification of tools for use in airborne software, both from a tool developer's and a tool user's perspective
- **DO-331 Model-Based Development and Verification**
  - addressing Model-Based Development (MBD) and verification and the ability to use modeling techniques to improve development and verification while avoiding pitfalls inherent in some modeling methods
- **DO-332 Object-Oriented Technology and Related Techniques**
  - addressing object-oriented software and the conditions under which it may be used
- **DO-333 Formal Methods**
  - addressing formal methods to complement (but not replace) testing

TQL-1 Qualification and DO-178C Objectives

- **TQL-1 ensures that …**
  - Tool Operational Requirements define all translation rules from model to the Source Code
  - Generated source code is
    - Compliant to a defined coding standard
    - Verifiable and Traceable
    - No unintended function or structure

- **DO-178C *Objectives* and TQL-1 qualification**
  - *Remove* some review/analysis objectives of generated code (A-5)
  - *Automate or remove* some low-level testing (A-6)
  - *Automate or remove* some structural coverage analysis on source (A-7)

  ➔ SAVE MONEY

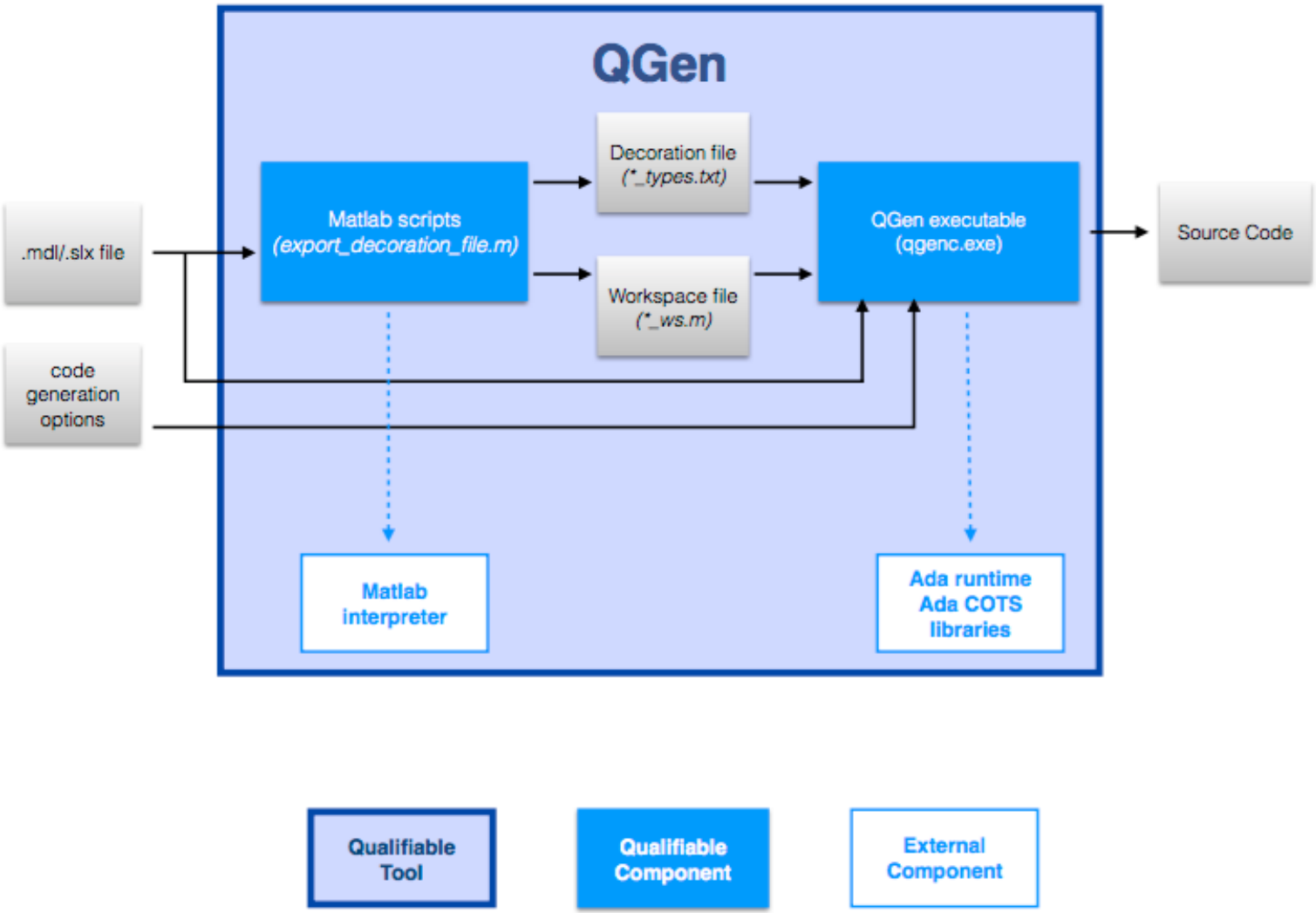
Source Code Verification Objectives: Shared Responsibility with User

- ***Tool Vendor:*** **TQL-1 Tool verification ensures correctness of the translation**
  - Tool Operational Requirements analysis and reviews
  - Code generated from models corresponds to that defined in TORs
    - Verify syntax and semantics of generated code comply with models
    - Equivalence between TORs and Simulink/Stateflow semantics is validated by the tool developer through simulation test cases
  - Source Code complies with coding standard defined in TORs

- ***Tool User:*** **TQL-1 Validation for project (DO-330, 6.2.1)**
  - *Qualifiable* vs. *Qualified*
  - "Tool Operational Requirements are sufficient and correct to eliminate, reduce, or automate the process(es) identified in the PSAC"
  - "The tool meets the needs of the software life cycle process in the *tool operational environment*"

TQL-1 Code Generator Architecture Overview

Qualifiable Code Generator architecture

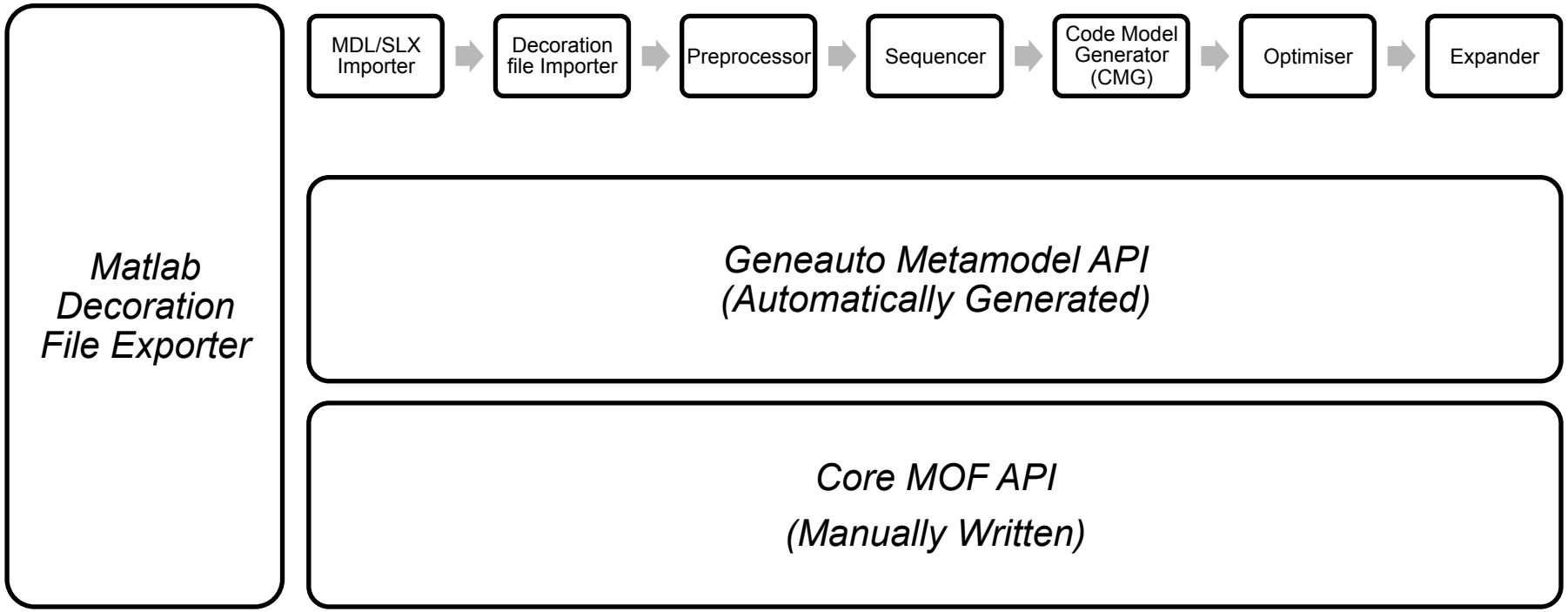


Detailed Code Generator architecture

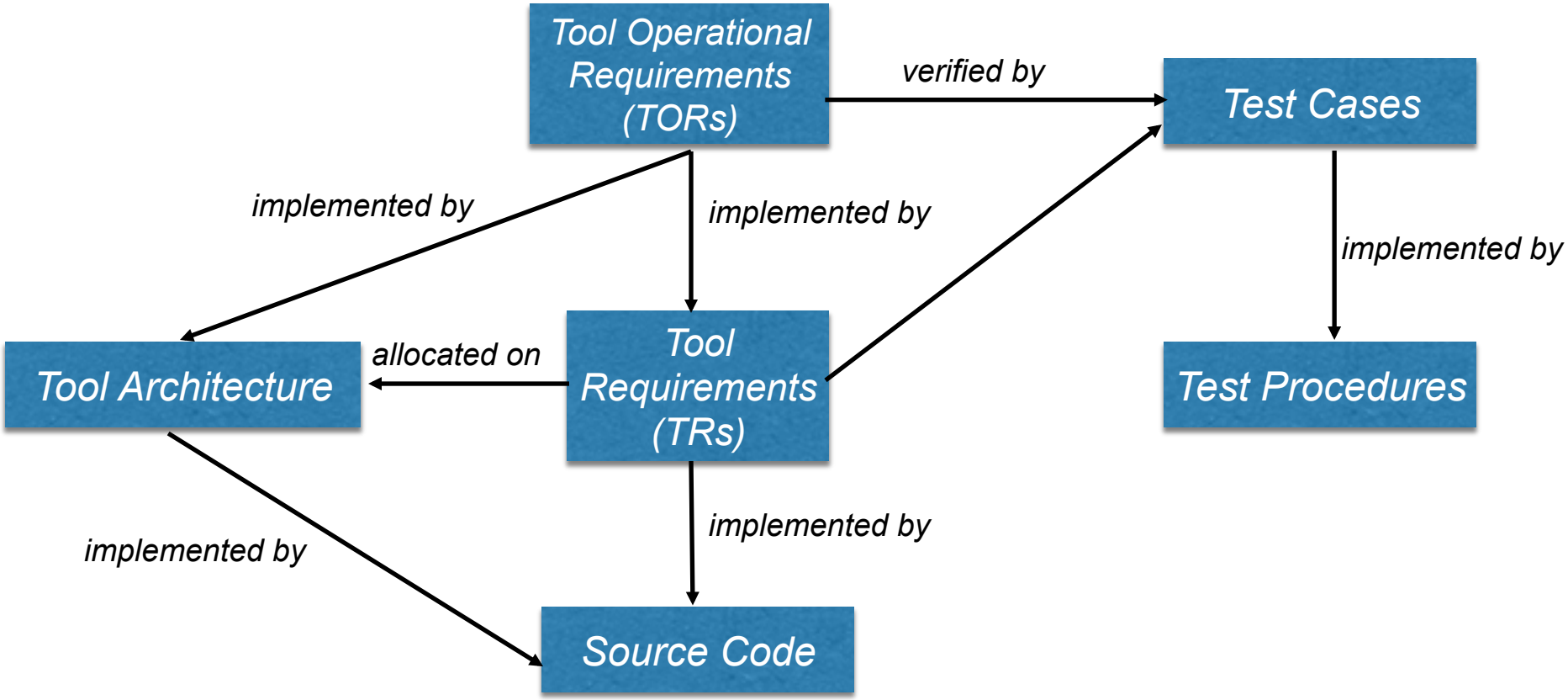
Overall Size:

- **50K SLOC *Handwritten***
- **50K SLOC *Generated from Metamodel***

I.e. *BIG* for highest level of assurance (think $50/SLOC)

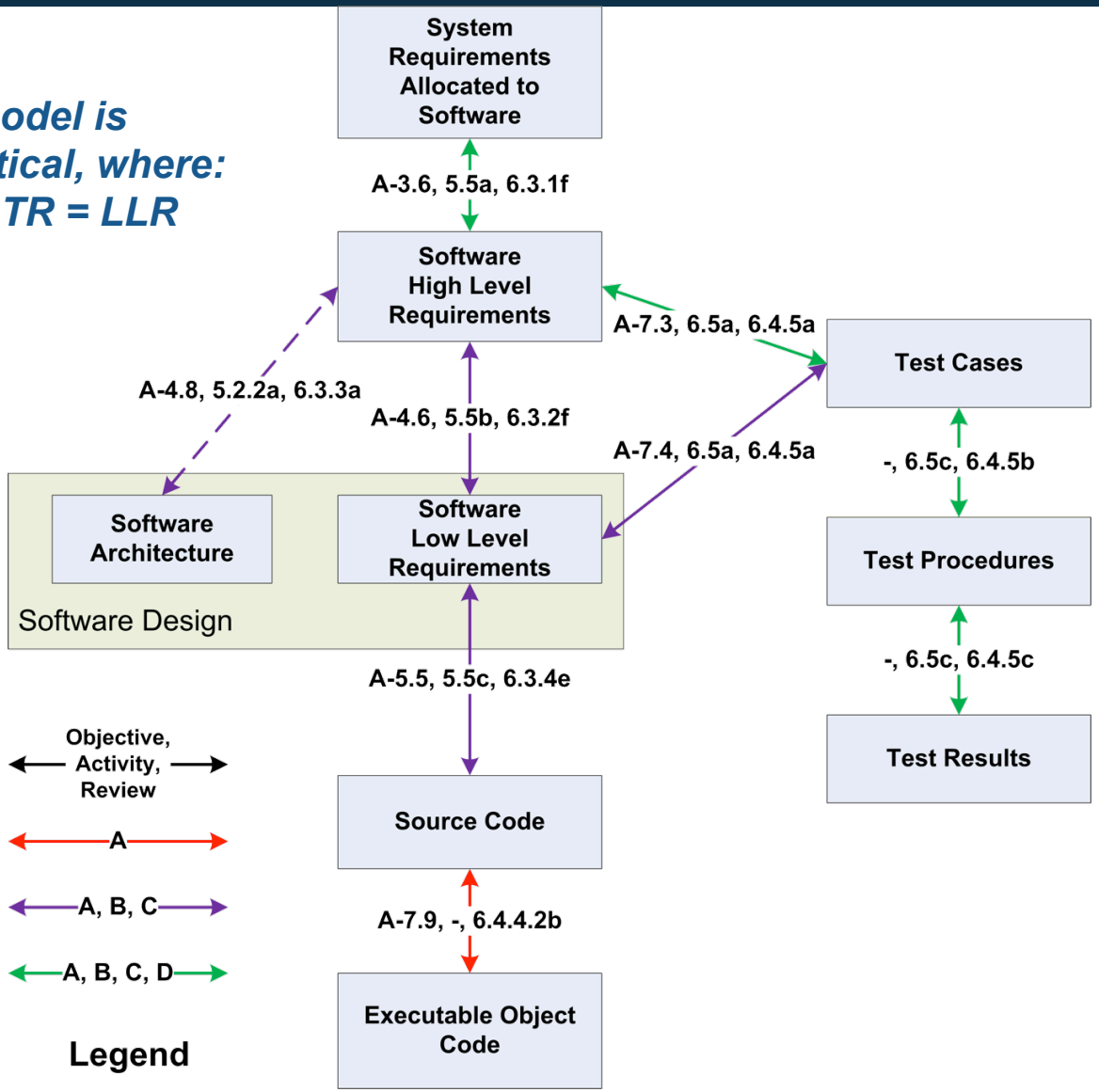


TQL1 Qualification: Requirements-Based Testing Model for Tools



DO-178C Requirements-Based Testing Model for Airborne System

TQL-1 testing model is essentially identical, where: TOR = HLR and TR = LLR



Approach for defining Tool Operational Requirements (TORs = HLRs)

- **TORs define transformations between input model and target language in steps**
  - TORs define the transformation between input model and intermediate language
    - Generic **functionality**
    - Generic parameters
    - Accepted combinations (types, dimensions, block parameters)

    TOR-Type-1
  - TORs to define transformations within the intermediate language
    - **Optimizations**
      - Removal of intermediate variables,…
    - **Expansions**
      - Typing, loops according to dimensions,…

    TOR-Type-2
  - TORs to define the representation in the target language (C or Ada)
    - Relation between generic and **concrete patterns**
    - Language-specific transformations
      - For example, use of specific ++ operator in C

    TOR-Type-3
- **TORs defining the transformation between the input model and the target language in a single step are too complex**
- **TORs are self-contained**
  - Defined semantics
  - Do not refer to Simulink/Stateflow documentation
    - Although obviously based on it
    - Tests verify that behavior of the generated source code is equivalent to the simulation

Intermediate P language

- **Goal**
  - Used to define abstract patterns of code
    - High-level language that abstracts aspects of dimensionality and concrete typing

- **Why**
  - Define clear semantics
    - Do not refer to Simulink/Stateflow documentation
  - Compact
  - Simplifies writing and understanding requirements
    - Remove details
  - Reduce combinatorial explosion
    - Pattern applies to every type (uint8, int8,…) and every dimension (scalar, vector, matrix)
      - And their combinations
  - Reuse among target languages
    - C, Ada,...

Tool Operational Requirements using intermediate P language

TOR-Type-1

- **Transformation between input model and intermediate P language**
  - Expected generic pattern of code
  - Generic parameters
    - Types of ports, types of parameters, values of parameters
  - Set of supported cases (combination of parameters)

TOR-Type-2

- **Transformations within the intermediate P language**
  - Source pattern → target pattern
  - We no longer have genericity on dimensions: scalar/vector/matrix have separate TORs
  - We still have genericity on operators and base types
  - No language-specific expansions

TOR-Type-3

- **Transformations between intermediate P language and target language (C or Ada)**
  - Regular printing rules
  - We are allowed to do intermediate expansion to account for language specific rules:
    - P language `a++;`
    - Ada-specific P language `a := a + 1;`

- **Exceptional blocks specified in natural language**
  - Virtual Subsystems that shall be inlined
  - Goto/From blocks

Example of TORs: Functional for Saturation (supported configuration)

TOR-Type-1

A legal Saturation block shall have 1 input, 1 output and the "UpperLimit" and "LowerLimit" parameter.
The base types of input port and output port shall coincide.
If the input and output are scalar then "UpperLimit" and "LowerLimit" shall be scalars.
If the input and output are vectors or matrices then "UpperLimit" and "LowerLimit" shall be either scalars or have the same dimensions as the block's output.
"UpperLimit" shall be greater than or equal to "LowerLimit".

TOR.1

Example of TORs: Functional for Saturation (scalar case)

TOR-Type-1

The Saturation block imposes upper and lower limits on an input signal. The block output is equal to the input value when the input value is within the range specified by the "LowerLimit" and "UpperLimit" parameters, "LowerLimit" when the input value is lower than "LowerLimit", and "UpperLimit" when the input value is greater than "UpperLimit".

if the block's output is scalar then the generated code model shall be:

```
if in < Params.LowerLimit
  out = Params.LowerLimit
else
  if in > Params.UpperLimit
    out = Params.UpperLimit
  else
    out = in
```

TOR.2

Example of TORs: Functional for Saturation (non scalar case)

TOR-Type-1

The Saturation block imposes upper and lower limits on an input signal. The block output is equal to the input value when the input value is within the range specified by the "LowerLimit" and "UpperLimit" parameters, "LowerLimit" when the input value is lower than "LowerLimit", and "UpperLimit" when the input value is greater than "UpperLimit".

if the block's output is a vector or a matrix and LowerLimit and UpperLimit are of the same dimension then the generated code model shall follow the pattern specified in **TOR.4** where:

Code_Fragment is:

```
if in < Params.LowerLimit
  out = Params.LowerLimit
else
  if in > Params.UpperLimit
    out = Params.UpperLimit
  else
    out = in
```

TOR.3

Dims are the dimensions of output port
Indexed_Expressions are: in, out, Params.LowerLimit, Params.UpperLimit

Example of TORs: Expansion

TOR-Type-2

TOR.4

The expansion of a code fragment for multidimensional data is parameterized by the following aspects:

- A code model fragment: *Code_Fragment*
- List of dimensions: *Dims*
- List of sub-expressions that are to be indexed in *Code_Fragment*: *Indexed_Expressions*

The following transformation shall be performed:

1. The code model fragment *Code_Fragment* shall be wrapped into N RangeIterationStatements, where N is the number of dimensions in *Dims*. N can only be 1 or 2.
2. The iteration ranges are 0 .. Dims (K) - 1, where K = 1 .. N
3. Each occurrence of an iterable expression in *Indexed_Expressions* shall be suffixed with M indices, where M is the dimensionality of the given *Indexed_Expression* and index values set to appropriate index variables
4. A literal non-scalar expression in *Indexed_Expressions* (block parameters only) shall be copied to a new variable named <block_name>_<parameter_Name>. Then, each occurrence of the given expression in *Indexed_Expressions* shall be replaced by the appropriate indexed variable expression

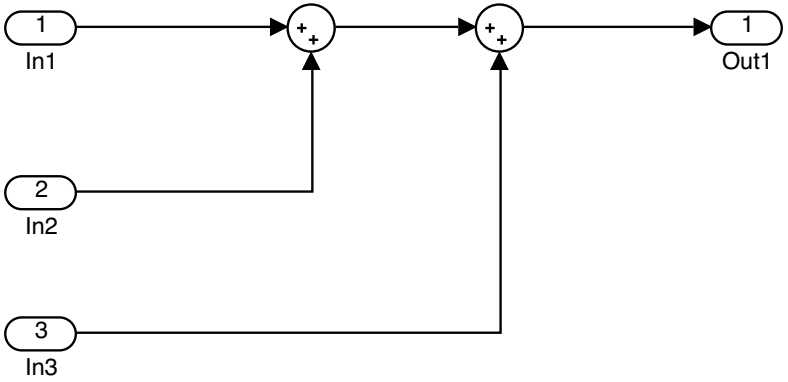
Example of TOR: Optimization

TOR-Type-2

A variable that is assigned but never read shall be removed **TOR.N**

A variable that is assigned once shall be removed if the following conditions are true:
- It is assigned in a simple assignment
- The assignment is not for a single element of a vector or a matrix
- The assignment is not for a component in a structure
- The variable is not the out argument of a subprogram call

TOR.N+1




```
Sum1_out1 = In1 + In2
Sum2_out1 = Sum1_out1 + In3
Out1 = Sum2_out1
```

TOR.N+1

```
Out1 = In1 + In2 + In3
```

Example of TOR: target code pattern

The C code generated for an if statement that contains an else alternative:

```
   if <condition>
      <if_statements>
   else
      <else_statements>
```

shall have the following pattern:

```
   if (<condition>) {
      <if_statements>
   } else {
      <else_statements>
   }
```

where:
<condition> is the controlling boolean expression of the if statement
<if_statements> is the sequence of statements if the condition evaluates to True
<else_statements> is the sequence of statements if the condition evaluates to False

TOR-Type-3

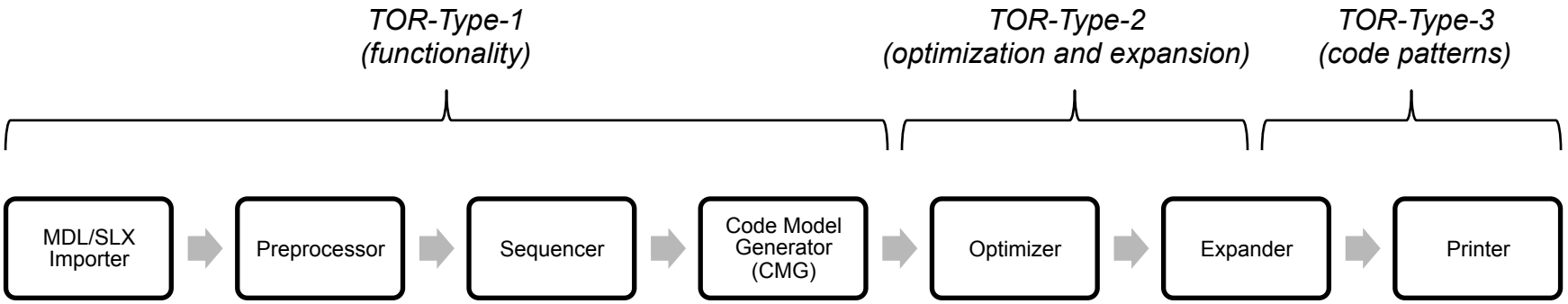
```
If A > 100
   Out1 = In1
else
   Out1 = In2
```

TOR.M

```
if (A > 100) {
   Out1 = In1;
} else {
   Out1 = In2;
}
```

TOR.M

Types of Tool Operational Requirements



- **Contains**
  - Specification of transformations
- **How**
  - MDL/SLX Importer
    - Natural language
  - Preprocessor
    - Natural language + Contracts
  - Sequencer
    - Natural language + Contracts
  - Code Model Generator
    - Natural language + pseudo-code in P language
  - Optimizer
    - Natural language
  - Expander
    - Natural language
  - Printer
    - Natural language

Example of Tool Requirements (TRs = LLRs): Configuration for “Saturate” block

- **TR.1.1 checks the configuration for two different Simulink blocks**
  - Saturation
  - Saturation Dynamic

The supported configuration of a Saturate block shall have: **TR.1.1**
- Either 3 inputs and 1 output, or 1 input, 1 output and the "UpperLimit" and "LowerLimit" parameters
- The base types of all input ports and output port shall coincide
- In the Saturation Dynamic block, "Saturate On Integer Overflow" flag shall be turned off (MISRA constraint)

If the input and output of a Saturation block are scalar then "UpperLimit" and "LowerLimit” shall be scalars. **TR.1.2**

If the input and output of a Saturation block are vectors or matrices then "UpperLimit" and "LowerLimit” shall be either scalars or have the same dimensions as the block’s output. **TR.1.3**

"UpperLimit” shall be greater than or equal to "LowerLimit”. **TR.1.4**

Requirement-Based Testing

- **Types of tests**
  - TOR-based test cases -- *High Level Requirements* based testing
  - TR-based test cases -- *Low Level Requirements* based testing

- **TOR-based test cases (DO-330, table T-6, obj. 1 & 2)**
  - Expected code from Simulink models
    - Input: Simulink models
    - Expected output: Check against expected **code patterns and behavior**
  - Combination of test cases with reduced and large number of blocks
  - One or more test cases per test procedure
  - Requirement-based testing

- **TR-based test cases (DO-330, table T-6, obj. 3 & 4)**
  - Check behavior based on internal model representation (at intermediate steps)
  - Expected behavior of subprogram
    - Input: cannot express as Simulink model
    - Output: cannot express as behavior of output code
  - Requirement-based testing based on internal context

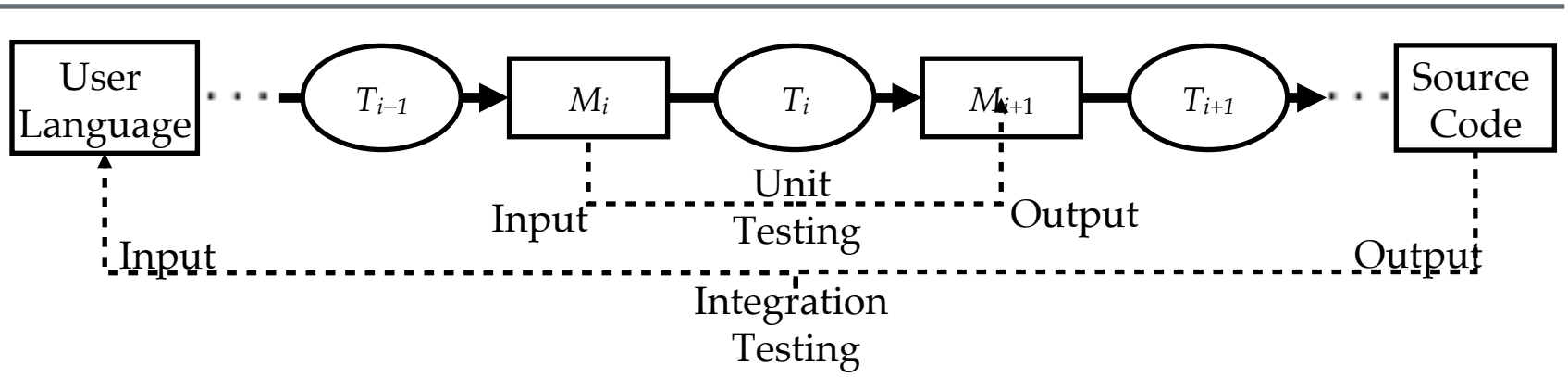
TOR-based (High-Level-Requirement-based) testing

- **Verify behavior of individual blocks**
  - Individual test cases for each block and each configuration
- **Verify optimization rules**
- **Verify generated code**
- **Verify composition of blocks**
  - Tester defines the expected result
  - Simulation used as oracle

TR-based test cases: Difficulties in performing unit testing of low-level requirements

- **Input**
  - Cannot express as Simulink model
  - Internal models are difficult to generate
- **Output**
  - Cannot express as behavior of output code
  - Generated internal models are difficult to verify
- **Traceability**
  - How to ensure that a concrete test case is exercised by a concrete test procedure
- **Coverage**
  - How to ensure that all test cases have been exercised
- **Expressiveness**
  - Compare state before and after the subprogram execution

Unit testing vs. Integration Testing



Integration testing is widely preferred to unit testing

	Unit Testing	Integration Testing
Test Data Editor/Viewer	Internal languages have no editors	User language has a good editor
Test Data Complexity	Intermediate languages are complex	User language is simpler, has higher abstraction
Test Exhaustiveness	Achievable thanks to isolation of units	Hard to achieve with no visibility on the internals of the tool

Objective

Achieve unit testing exhaustiveness using only integration tests

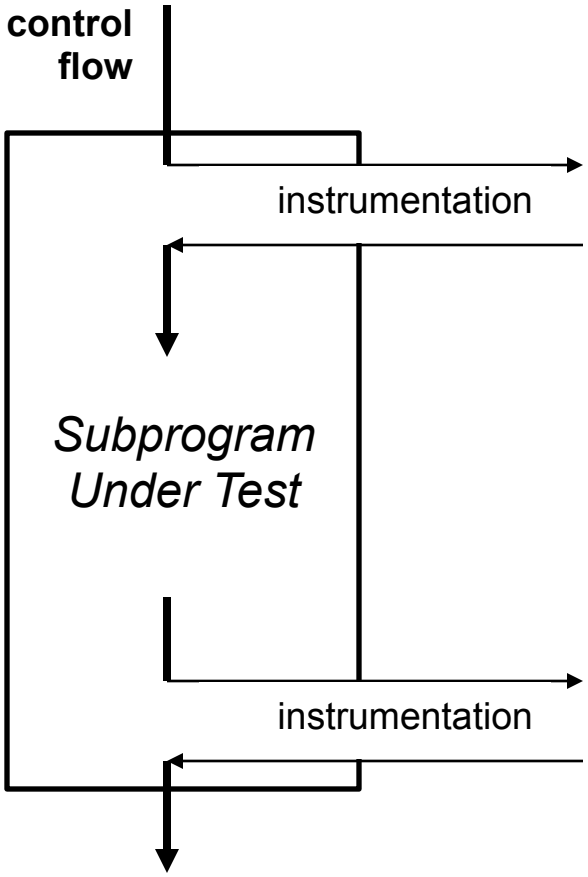
TR-based test cases: Approach to handling low-level requirements “unit” testing

- **We need to define:**
  - Equivalence class definition
    - Characterization of the input relevant to the test case
  - Oracle
    - Output validity criteria
    - The way to decide the pass/fail verdict

- **Test procedures can be:**
  - End-to-end (from Simulink to code) tests
    - Covering a set of test cases at different phases
  - Unit tests
    - Covering a set of test cases for a given TR

- **Executable expression of test cases**
  - Instrument code of the tool, so at execution time we can
    - Log the occurrence of test cases (traceability and coverage)
    - Check (test oracle) the expected result of the test case
  - Tool does not have instrumentation in production mode
  - Equivalence between instrumented and non-instrumented version
    - Execute the test procedures in both modes and check that generated code is exactly the same

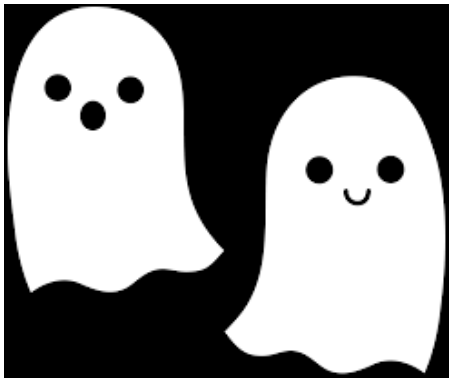
Instrumentation for Test Case Evaluation



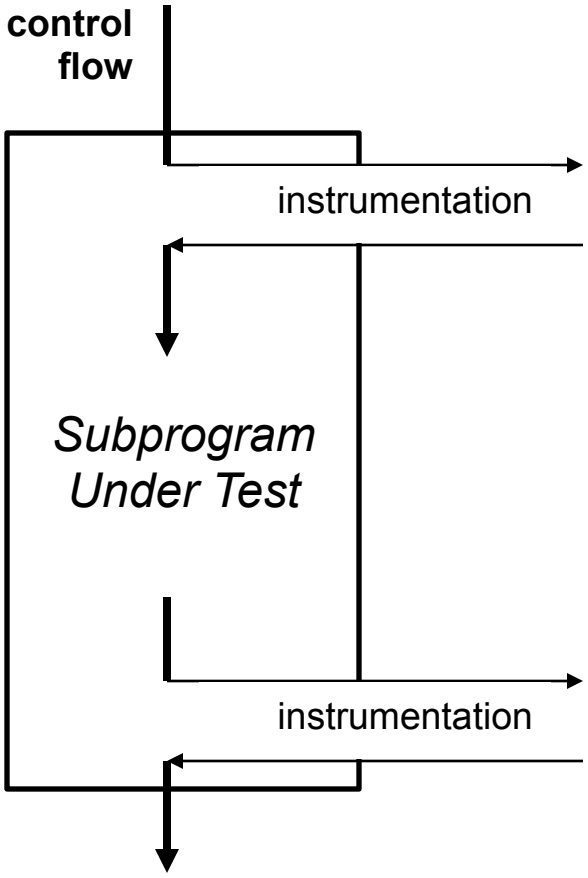
Log which equivalence classes are covered by the input data

Evaluate the test oracles of the covered test cases

All *instrumentation* is marked as *ghost* code
Compiler produces ghost code only when assertions are enabled.



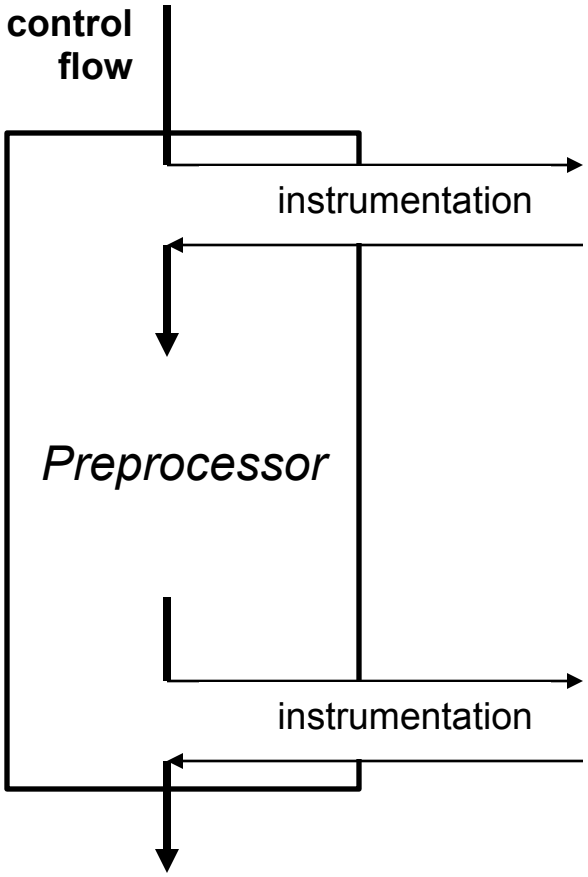
Instrumentation for Test Case Evaluation




```
for TC of Test_Cases loop
   if TC.Equivalence_Class then
      TC.Covered := True;
      Log (TC.Id & " Covered");
   end if;
end loop;
```

```
for TC of Test_Cases loop
   if TC.Covered then
      if not TC.Oracle then
         Log (TC.Id & " Failed");
      end if;
   end if;
end loop;
```

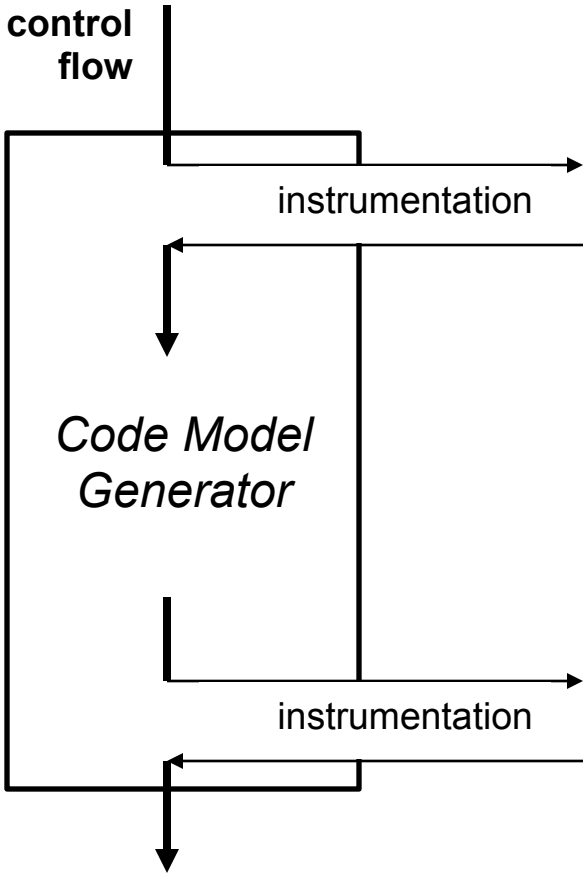
Instrumentation for Test Case Evaluation




```
function Equivalence_Class (T : Process_Goto_From_TC1)
                                                  return Boolean is
-- Input model contains "Goto" blocks
   Goto_Blocks_In_Model : Element_Sequence'Class :=
        Model.Get_Blocks_Of_Type ("Goto");
begin
   if Goto_Blocks_In_Model.Length > 0 then
      T.Goto_Blocks.Append (Goto_Blocks_In_Model);
      return True;
   else
      return False;
   end if;
end Equivalence_Class;
```

```
function Oracle (T : Process_Goto_From_TC1)
                                                  return Boolean is
-- Check that all "Goto" blocks become null and are no longer
-- contained in the model
begin
   return (for all B of T.Goto_Blocks =>
             B.Is_Null and then
               not Model.Contained_Elements.Contains (B));
end Oracle;
```

Instrumentation for Test Case Evaluation




```
function Equivalence_Class (T : TC1) return Boolean is
  (B.Inports (1).Is_Vector and then B.UpperLimit.Is_Scalar);

function Equivalence_Class (T : TC2) return Boolean is
  (B.Inports (1).Is_Vector and then B.UpperLimit.Is_Vector);

function Equivalence_Class (T : TC3) return Boolean is
  (B.Inports (1).Is_Matrix and then B.UpperLimit.Is_Matrix
     and then (for some V in B.UpperLimit =>
                  V.Is_Infinity));

...
```

Test Oracle does not have to be automatic. It might be:

- Printing the intermediate code model, validating it and setting it as a test baseline
- Validating the final generated code because it has a structure very similar to the code model, and setting it as a test baseline

TR-based test cases: Implementation

- **Test cases are expressed as:**
  - Executable form
    - Test objective =>
      - Evaluated at subprogram entry
    - Test oracles =>
      - Evaluated at subprogram exit
  - Manual verifications
    - If applying the previous are difficult

```
function Sqrt (X : Integer) return Integer with
  Test_Case =>
    (Name => "TC1", Mode => Nominal,
     Requires => X = 1,
     Ensures  => Sqrt'Result = 1),
  Test_Case =>
    (Name => "TC2", Mode => Nominal,
     Requires => X > 1,
     Ensures  => Sqrt'Result ** 2 <= X
                    and then
                 (Sqrt'Result + 1) ** 2 > X);
```

- **Traceability and coverage**
  - Logging occurrences of requires and ensures
  - Requires
    - Evidences of coverage of test cases
  - Ensures
    - Test case results

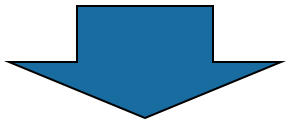
```
Trace for a test with Sqrt (5):

Sqrt.TC1.Requires      False
Sqrt.TC2.Requires      True
Sqrt.TC1.Ensures       Not Applicable
Sqrt.TC2.Ensures       True
```

TR-based test cases: Implementation (II)

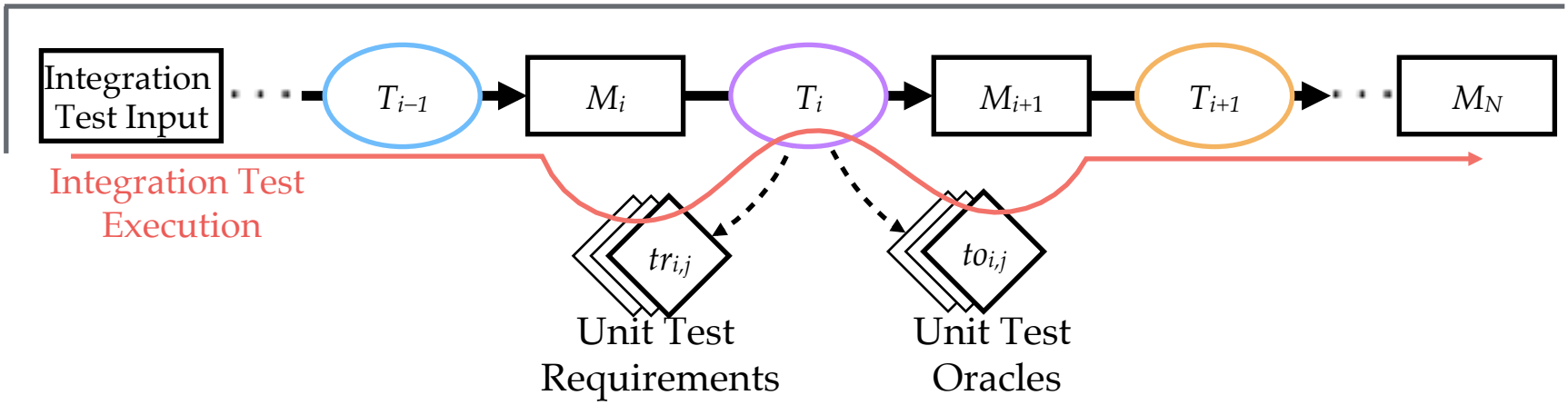
- **Some flexibility**
  - Instead of Test_Case we can use Contract_Case, Precondition, Postcondition
- **Expressiveness**
  - We can save and check state using a separate verification package
- **Use "Ghost" Code throughout**
  - Ghost code removed when compiled in no-assertions mode
- **Example using Executable Preconditions/Postconditions:**

```
Procedure to eliminate GoTo blocks. Test case is:
 - Requires: There exists a GoTo block
 - Ensures: That GoTo block becomes null and is removed from the model
```



```
procedure Preprocess (SB : SystemBlock'Class) with
   Pre  => Preprocess_Verif.Preprocess_Test_Cases_Requires (SB),
   Post => Preprocess_Verif.Preprocess_Test_Cases_Ensures (SB);
--  Routines called in the pre- and post-conditions can store any required data within
--  internal variables and check their status after execution of Preprocess
```

Integrated Unit Testing



	Unit Test Requirements													
Integration Tests	$tr_{0,0}$	$tr_{0,1}$	$tr_{0,2}$	$to_{0,0}$	$to_{0,1}$	$to_{0,2}$	$tr_{1,0}$	$tr_{1,1}$	$to_{1,0}$	$to_{1,1}$	$tr_{2,0}$	$tr_{2,1}$	$to_{2,0}$	$to_{2,1}$
$Test_0$	SAT	–	–	PASS	–	–	–	SAT	–	PASS	–	SAT	–	PASS
$Test_1$	–	–	SAT	–	–	PASS	–	SAT	–	PASS	–	SAT	–	PASS
$Test_2$	–	–	SAT	–	–	FAIL	SAT	–	PASS	–	–	SAT	–	PASS
$Test_3$	–	–	–	–	–	–	SAT	SAT	PASS	PASS	–	SAT	–	FAIL
$Test_4$	SAT	–	–	PASS	–	–	–	SAT	–	PASS	–	SAT	–	PASS

Non-covered Unit Test Case

Unit Test Failure

Non-covered Unit Test Case

Unit Test Failure

TQL-1 Qualification Approach – Building *Trust* in Code Generator

- **Structure of Tool (Operational) Requirements reflects Multi-Pass structure of Code Generator**
  - Input, Transformation(s), Optimization(s), Output
- **Conventional Unit Testing of Multi-Pass Code Generator is painful**
  - Difficult to create inputs and check outputs of individual passes
- **Integration Testing is generally easier**
  - can use normal Models as Input; generated Source Code as output
- **Instrument Code Generator with "ghost" code to use Integration Testing to accomplish Unit Testing**
  - *Monitor* inputs during Integration Testing to be sure all relevant Test Cases go through each pass
  - Can check *completeness* and *uniqueness* of input classification, so each input gets exactly one classification
  - Use Oracle to create *expected* output based on input classification and check for *match*
- ***Coverage* Testing now at Test Case level**
  - Verify that all interesting test cases were executed
- **Approach applicable to other multi-pass tools such as Compilers**