Implementation of an Interpreter for the Test Purpose Specification Language TDL^{TP}

Tanel Prikk

Supervisor: Jüri Vain, PhD

Co-supervisor: Evelin Halling, PhD

Tallinn University of Technology

School of Information Technologies

2019

Presentation Outline

1. Context
2. Objective
3. Implementation
4. Validation
5. Summary

Next Section

1. Context
    - Model-Based Testing
    - UPPAAL
    - Test Purpose Specification Language TDL$^{\mathrm{TP}}$
2. Objective
3. Implementation
4. Validation
5. Summary

Context – Model-Based Testing

Model-Based Testing (MBT)

Black-box testing method where explicit models derived from requirements are used to verify the correctness of a system under test (SUT).

Context – Model-Based Testing

Model-Based Testing (MBT)

Black-box testing method where explicit models derived from requirements are used to verify the correctness of a system under test (SUT).

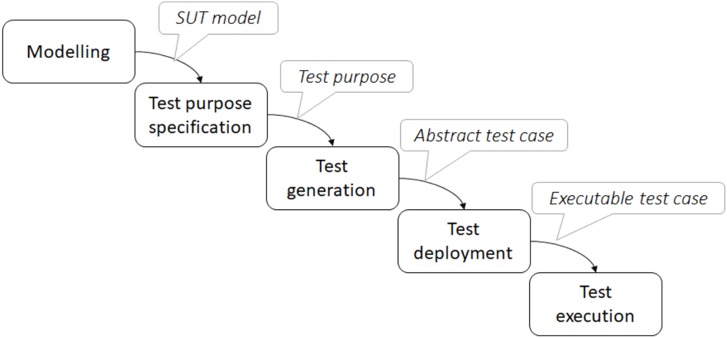


Figure 1. Potential MBT workflow.

Context – Model-Based Testing

Model-Based Testing (MBT)

Black-box testing method where explicit models derived from requirements are used to verify the correctness of a system under test (SUT).

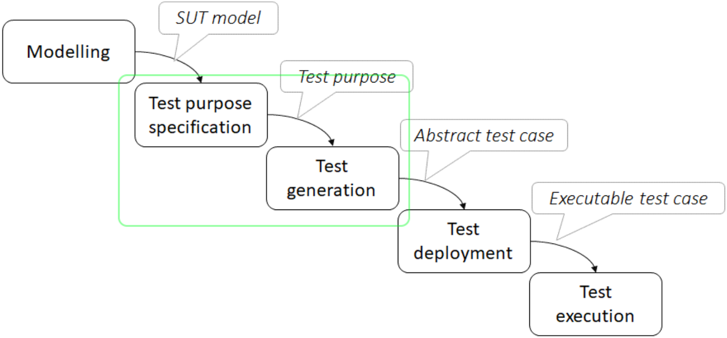


Figure 1. Potential MBT workflow.

Context – Model-Based Testing (cont'd)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Context – Model-Based Testing (cont'd)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Requirements for automation:

Context – Model-Based Testing (cont'd)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Requirements for automation:

- notation for expressing test purposes;

Context – Model-Based Testing (cont'd)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Requirements for automation:

- notation for expressing test purposes;
- machine-interpretable modelling formalism.

Context – Model-Based Testing (cont’d)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Requirements for automation:

- notation for expressing test purposes;
- machine-interpretable modelling formalism.

Benefits of automation:

Context – Model-Based Testing (cont'd)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Requirements for automation:

- notation for expressing test purposes;
- machine-interpretable modelling formalism.

Benefits of automation:

- eliminate redundant activities;

Context – Model-Based Testing (cont'd)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Requirements for automation:

- notation for expressing test purposes;
- machine-interpretable modelling formalism.

Benefits of automation:

- eliminate redundant activities;
- prevent human errors;

Context – Model-Based Testing (cont'd)

Test Generation

Derivation of abstract test sequences from the state space of a SUT model using test purposes as guiding constraints.

Requirements for automation:

- notation for expressing test purposes;
- machine-interpretable modelling formalism.

Benefits of automation:

- eliminate redundant activities;
- prevent human errors;
- reduce costs.

Context – UPPAAL

UPPAAL Timed Automata (UTA)

Modeling formalism where systems are represented as networks of state transition graphs annotated with timing constraints.

Context – UPPAAL

UPPAAL Timed Automata (UTA)

Modeling formalism where systems are represented as networks of state transition graphs annotated with timing constraints.

Facilitates modeling of real-time systems.

Context – UPPAAL

UPPAAL Timed Automata (UTA)

Modeling formalism where systems are represented as networks of state transition graphs annotated with timing constraints.

Facilitates modeling of real-time systems.

The UPPAAL toolkit includes the following core tools:

Context – UPPAAL

UPPAAL Timed Automata (UTA)

Modeling formalism where systems are represented as networks of state transition graphs annotated with timing constraints.

Facilitates modeling of real-time systems.

The UPPAAL toolkit includes the following core tools:

- graphical environment for defining UTA models,

Context – UPPAAL

UPPAAL Timed Automata (UTA)

Modeling formalism where systems are represented as networks of state transition graphs annotated with timing constraints.

Facilitates modeling of real-time systems.

The UPPAAL toolkit includes the following core tools:

- graphical environment for defining UTA models,
- simulator which allows user to execute a model and observe its behavior,

Context – UPPAAL

UPPAAL Timed Automata (UTA)

Modeling formalism where systems are represented as networks of state transition graphs annotated with timing constraints.

Facilitates modeling of real-time systems.

The UPPAAL toolkit includes the following core tools:

- graphical environment for defining UTA models,
- simulator which allows user to execute a model and observe its behavior,
- model-checker (`verifyta`) – provides tools for the formal verification of correctness properties for the model (and the generation of *witness traces* which prove these properties).

Context – UPPAAL (cont'd)

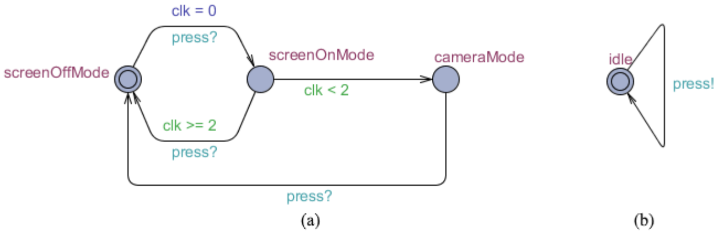


Figure 2. Example UPPAAL automata: (a) smartphone, (b) user.

Context – UPPAAL (cont'd)

UPPAAL's property specification language can be used to express test purposes.

Context – UPPAAL (cont'd)

UPPAAL's property specification language can be used to express test purposes.

Formula types:

Context – UPPAAL (cont'd)

UPPAAL's property specification language can be used to express test purposes.

Formula types:

- state formulae:

  `systemProcess.workCompleted;`

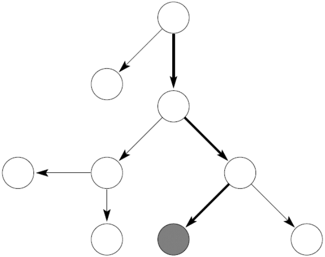


Figure 3. Verification of reachability property `E<>p`.

Context – UPPAAL (cont'd)

UPPAAL's property specification language can be used to express test purposes.

Formula types:

- state formulae:

  `systemProcess.workCompleted`;
- path formulae:

  `E<>systemProcess.workCompleted`.

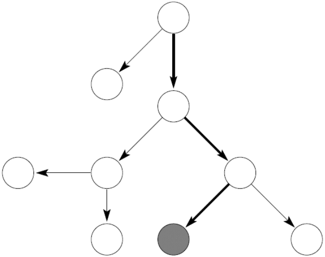


Figure 3. Verification of reachability property `E<>p`.

Context – Test Purpose Specification Language TDL^{TP}

Context – Test Purpose Specification Language TDL^{TP}

Limitations of UPPAAL test purpose specification language:

Context – Test Purpose Specification Language TDL$^{\mathrm{TP}}$

Limitations of UPPAAL test purpose specification language:

- temporal operators cannot be nested;

Context – Test Purpose Specification Language TDL^{TP}

Limitations of UPPAAL test purpose specification language:

- temporal operators cannot be nested;
- complex test purposes require manual modification of SUT model;

Context – Test Purpose Specification Language TDL^{TP}

Limitations of UPPAAL test purpose specification language:

- temporal operators cannot be nested;
- complex test purposes require manual modification of SUT model;
- UPPAAL MBT only feasible for expert users.

Context – Test Purpose Specification Language TDL^{TP}

Limitations of UPPAAL test purpose specification language:

- temporal operators cannot be nested;
- complex test purposes require manual modification of SUT model;
- UPPAAL MBT only feasible for expert users.

Solution:

Evelin Halling, Jüri Vain, Artem Boyarchuk, and Oleg Illiashenko, **"Test Scenario Specification Language for Model-based Testing"**, *International Journal of Computing*, 2019.

Context – Test Purpose Spec. Language TDL$^{\text{TP}}$ (cont'd)

In TDL$^{\text{TP}}$:

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)

In TDL^{TP}:

- test purposes are encoded as logical expressions, e.g.

```
A(TS1) ~> E(TS2) & E(TS3);
```

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)

In TDL^{TP}:

- test purposes are encoded as logical expressions, e.g. `A(TS1) ~> E(TS2) & E(TS3)`;
- ground level terms in the expression are references to sets of transitions in the SUT model;

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)

In TDL^{TP}:

- test purposes are encoded as logical expressions, e.g.
  `A(TS1) ~> E(TS2) & E(TS3);`
- ground level terms in the expression are references to sets of transitions in the SUT model;
- once expression is interpreted, a behaviorally equivalent test model is produced;

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)

In TDL^{TP}:

- test purposes are encoded as logical expressions, e.g.
  `A(TS1) ~> E(TS2) & E(TS3);`
- ground level terms in the expression are references to sets of transitions in the SUT model;
- once expression is interpreted, a behaviorally equivalent test model is produced;
- test model is an extension of SUT model which contains a *property recognizer automata* tree that structurally mirrors the abstract syntax tree (AST) of the expression;

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)

In TDL^{TP}:

- test purposes are encoded as logical expressions, e.g. `A(TS1) ~> E(TS2) & E(TS3);`
- ground level terms in the expression are references to sets of transitions in the SUT model;
- once expression is interpreted, a behaviorally equivalent test model is produced;
- test model is an extension of SUT model which contains a *property recognizer automata* tree that structurally mirrors the abstract syntax tree (AST) of the expression;
- root automaton in the recognizer tree (`stopwatch`) is intended to be used in the UPPAAL reachability query `E<>stopwatch.pass`;

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)

In TDL^{TP}:

- test purposes are encoded as logical expressions, e.g. `A(TS1) ~> E(TS2) & E(TS3);`
- ground level terms in the expression are references to sets of transitions in the SUT model;
- once expression is interpreted, a behaviorally equivalent test model is produced;
- test model is an extension of SUT model which contains a *property recognizer automata* tree that structurally mirrors the abstract syntax tree (AST) of the expression;
- root automaton in the recognizer tree (`stopwatch`) is intended to be used in the UPPAAL reachability query `E<>stopwatch.pass`;
- witness trace generated per query is an abstract test case.

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)

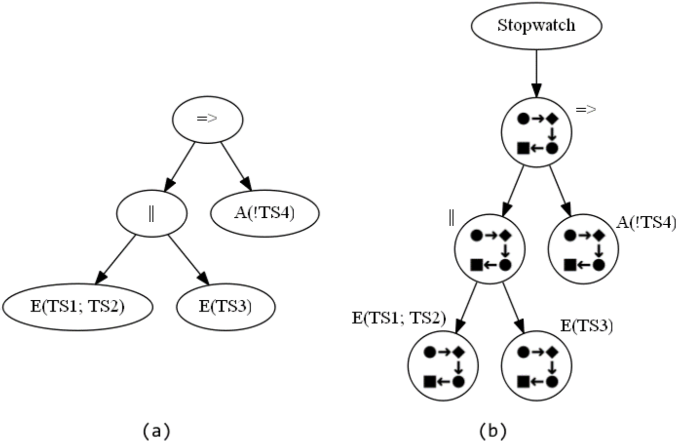
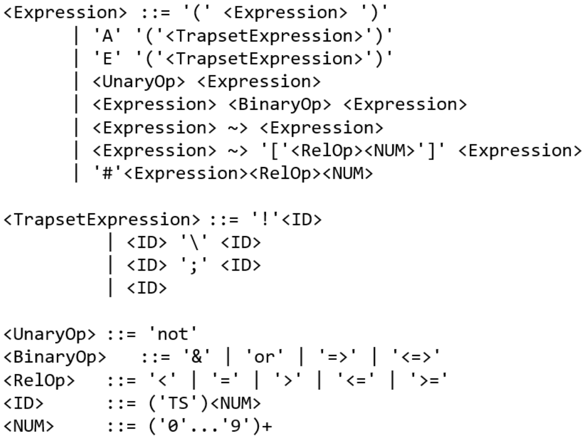


Figure 4. Comparison of TDL^{TP} AST (a) and corresponding recognizer tree (b).

Context – Test Purpose Spec. Language TDL^{TP} (cont'd)



```
<Expression> ::= '(' <Expression> ')'
       | 'A' '('<TrapsetExpression>')'
       | 'E' '('<TrapsetExpression>')'
       | <UnaryOp> <Expression>
       | <Expression> <BinaryOp> <Expression>
       | <Expression> ~> <Expression>
       | <Expression> ~> '['<RelOp><NUM>']' <Expression>
       | '#'<Expression><RelOp><NUM>

<TrapsetExpression> ::= '!'<ID>
          | <ID> '\' <ID>
          | <ID> ';' <ID>
          | <ID>

<UnaryOp> ::= 'not'
<BinaryOp>   ::= '&' | 'or' | '=>' | '<=>'
<RelOp>   ::= '<' | '=' | '>' | '<=' | '>='
<ID>      ::= ('TS')<NUM>
<NUM>     ::= ('0'...'9')+
```

Figure 5. TDL^{TP} grammar.

Next Section

1 Context

2 Objective

3 Implementation

4 Validation

5 Summary

Objective

Goal Statement

Implement an interpreter that accepts as input:

- a $\mathrm{TDL}^{\mathrm{TP}}$ expression,
- a UPPAAL SUT model,

and produces a **test model** as output.

Objective

Goal Statement

Implement an interpreter that accepts as input:

- a TDL$^{\mathrm{TP}}$ expression,
- a UPPAAL SUT model,

and produces a **test model** as output.

Caveats:

Objective

Goal Statement

Implement an interpreter that accepts as input:

- a TDL$^{\text{TP}}$ expression,
- a UPPAAL SUT model,

and produces a **test model** as output.

Caveats:

- efficiency is not an immediate concern;

Objective

Goal Statement

Implement an interpreter that accepts as input:

- a TDL$^{\text{TP}}$ expression,
- a UPPAAL SUT model,

and produces a **test model** as output.

Caveats:

- efficiency is not an immediate concern;
- assume model validity wrt system requirements.

Objective (cont'd)

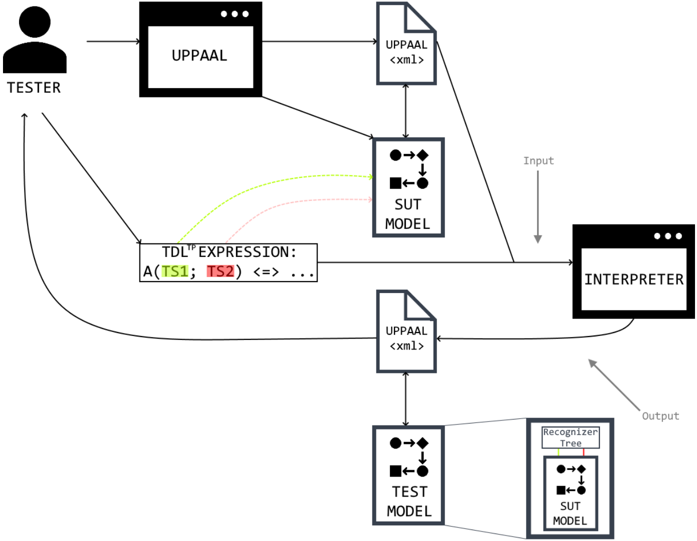


Figure 6. Interaction diagram.

Next Section

1. Context
2. Objective
3. Implementation
   - Approach
   - Structure
   - Demonstration
4. Validation
5. Summary

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;
- design initial component layout;

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;
- design initial component layout;
- implement bottom-up;

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;
- design initial component layout;
- implement bottom-up;
- adjust when appropriate.

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;
- design initial component layout;
- implement bottom-up;
- adjust when appropriate.

Benefits:

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;
- design initial component layout;
- implement bottom-up;
- adjust when appropriate.

Benefits:

- limit coupling;

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;
- design initial component layout;
- implement bottom-up;
- adjust when appropriate.

Benefits:

- limit coupling;
- enforce reusability;

Implementation – Approach

Takes inspiration from Component-Based Development (CBD).

General process:

- identify necessary sub-functionalities;
- design initial component layout;
- implement bottom-up;
- adjust when appropriate.

Benefits:

- limit coupling;
- enforce reusability;
- facilitate refactoring.

Implementation – Structure

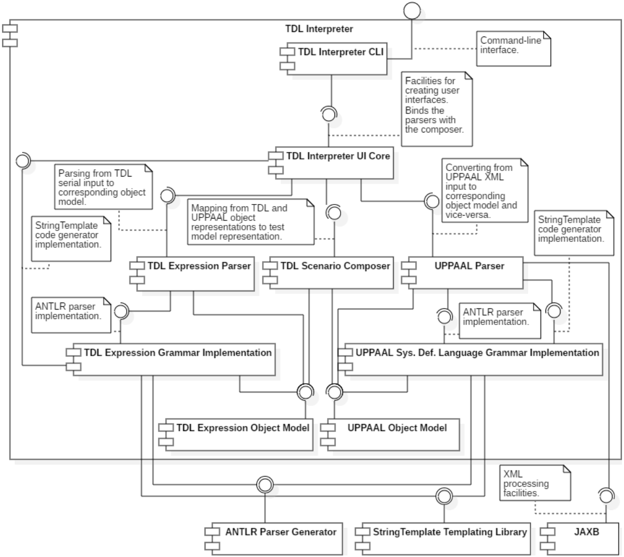
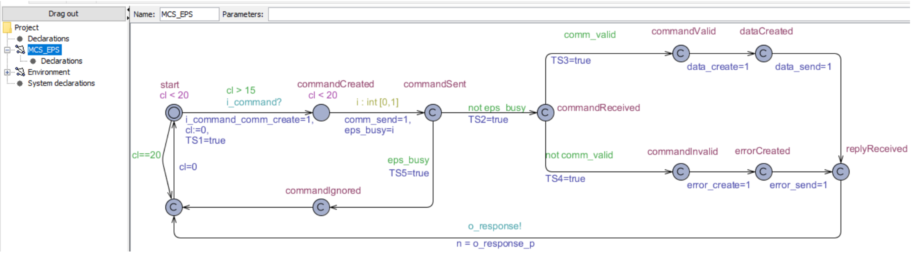

TDL Interpreter
TDL Interpreter CLI
Command-line interface.
Facilities for creating user interfaces. Binds the parsers with the composer.
TDL Interpreter UI Core
Parsing from TDL serial input to corresponding object model.
Converting from UPPAAL XML input to corresponding object model and vice-versa.
Mapping from TDL and UPPAAL object representations to test model representation.
StringTemplate code generator implementation.
StringTemplate code generator implementation.
TDL Expression Parser
TDL Scenario Composer
UPPAAL Parser
ANTLR parser implementation.
ANTLR parser implementation.
TDL Expression Grammar Implementation
UPPAAL Sys. Def. Language Grammar Implementation
TDL Expression Object Model
UPPAAL Object Model
XML processing facilities.
ANTLR Parser Generator
StringTemplate Templating Library
JAXB

Figure 7. Partial component diagram.

Demonstration

Demonstration

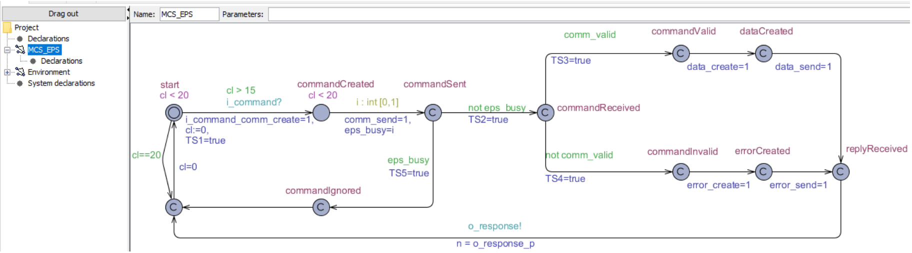


Name: MCS_EPS

Parameters:

Figure 8. Example UPPAAL input model.

Demonstration



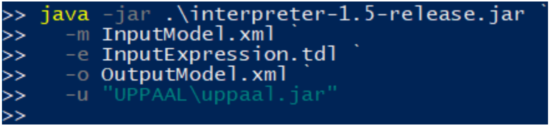
Name: MCS_EPS Parameters:

Figure 8. Example UPPAAL input model.

```
A(TS2;TS4) ~> E(TS2;TS3)
```

Figure 9. Example TDL^{TP} input expression.

Demonstration (cont'd)



```
>> java -jar .\interpreter-1.5-release.jar `
>>    -m InputModel.xml `
>>    -e InputExpression.tdl `
>>    -o OutputModel.xml `
>>    -u "UPPAAL\uppaal.jar"
>>
```

Figure 10. Call to interpreter command-line interface.

Demonstration (cont'd)

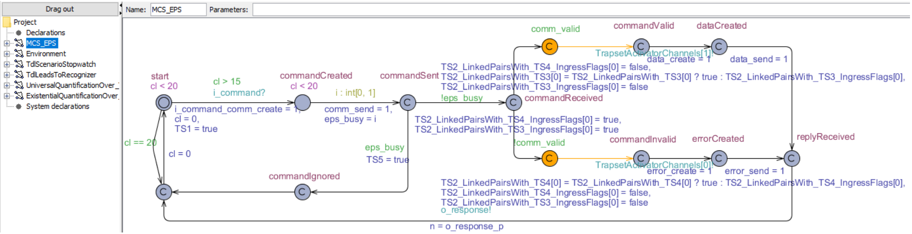

Drag out
Project
Declarations
MCS_EPS
Environment
TdlScenarioStopwatch
TdlLeadsToRecognizer
UniversalQuantificationOver_
ExistentialQuantificationOver_
System declarations
Name: MCS_EPS
Parameters:
start
cl < 20
cl > 15
i_command?
i_command_comm_create = 1,
cl = 0,
TS1 = true
commandCreated
cl < 20
i : int[0, 1]
comm_send = 1,
eps_busy = i
commandSent
!eps_busy
eps_busy
TS5 = true
cl == 20
cl = 0
commandIgnored
commandReceived
TS2_LinkedPairsWith_TS4_IngressFlags[0] = false,
TS2_LinkedPairsWith_TS3[0] = TS2_LinkedPairsWith_TS3[0] ? true : TS2_LinkedPairsWith_TS3_IngressFlags[0],
TS2_LinkedPairsWith_TS3_IngressFlags[0] = false
TS2_LinkedPairsWith_TS4_IngressFlags[0] = true,
TS2_LinkedPairsWith_TS3_IngressFlags[0] = true
comm_valid
commandValid
dataCreated
TrapsetActivatorChannels[1]!
data_create = 1
data_send = 1
!comm_valid
commandInvalid
errorCreated
replyReceived
TrapsetActivatorChannels[0]!
error_create = 1
error_send = 1
TS2_LinkedPairsWith_TS4[0] = TS2_LinkedPairsWith_TS4[0] ? true : TS2_LinkedPairsWith_TS4_IngressFlags[0],
TS2_LinkedPairsWith_TS4_IngressFlags[0] = false,
TS2_LinkedPairsWith_TS3_IngressFlags[0] = false
o_response!
n = o_response_p

Figure 11. Example test model.

Next Section

1 Context

2 Objective

3 Implementation

4 Validation

5 Summary

Validation

Observations:

Validation

Observations:

- Multiple component interactions are involved in test model generation – need to ensure the output conforms to rules of TDL^{TP}.

Validation

Observations:

- Multiple component interactions are involved in test model generation – need to ensure the output conforms to rules of TDL^{TP}.
- Rules are well-defined, so automated validation is feasible (but complicated).

Validation

Observations:

- Multiple component interactions are involved in test model generation – need to ensure the output conforms to rules of TDL^{TP}.
- Rules are well-defined, so automated validation is feasible (but complicated).
- Due to time limitations, most validation efforts were manual.

Validation

Observations:

- Multiple component interactions are involved in test model generation – need to ensure the output conforms to rules of TDL^{TP}.
- Rules are well-defined, so automated validation is feasible (but complicated).
- Due to time limitations, most validation efforts were manual.
- User input processing partially covered by automated tests.

Validation (cont'd)

Automated tests for language parsers:

Validation (cont'd)

Automated tests for language parsers:

- *Strategy*: structural validation of abstract syntax tree (AST).

Validation (cont'd)

Automated tests for language parsers:

- *Strategy*: structural validation of abstract syntax tree (AST).
- *Approach*: s-expressions (notation for representing tree structures) as common intermediate format.

Validation (cont'd)

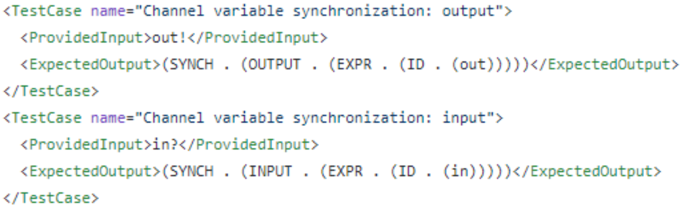
Automated tests for language parsers:

- *Strategy*: structural validation of abstract syntax tree (AST).
- *Approach*: s-expressions (notation for representing tree structures) as common intermediate format.
- 109 automated test cases – handful of programming defects discovered and resolved.

Validation (cont'd)

Automated tests for language parsers:

- *Strategy*: structural validation of abstract syntax tree (AST).
- *Approach*: s-expressions (notation for representing tree structures) as common intermediate format.
- 109 automated test cases – handful of programming defects discovered and resolved.



```
<TestCase name="Channel variable synchronization: output">
  <ProvidedInput>out!</ProvidedInput>
  <ExpectedOutput>(SYNCH . (OUTPUT . (EXPR . (ID . (out)))))</ExpectedOutput>
</TestCase>
<TestCase name="Channel variable synchronization: input">
  <ProvidedInput>in?</ProvidedInput>
  <ExpectedOutput>(SYNCH . (INPUT . (EXPR . (ID . (in)))))</ExpectedOutput>
</TestCase>
```

Figure 12. Example automated UPPAAL language parser test cases.

Validation (cont'd)

Manual integration tests:

- *Strategy*: define basic input model, TDL$^{\text{TP}}$ expression.

Validation (cont'd)

Manual integration tests:

- *Strategy*: define basic input model, TDL^{TP} expression.
- *Approach*: apply expression to model using interpreter, validate visually.

Validation (cont'd)

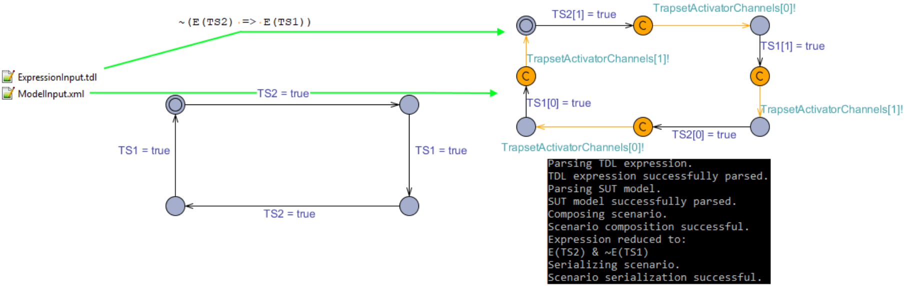
Manual integration tests:

- *Strategy*: define basic input model, TDL$^{\mathrm{TP}}$ expression.
- *Approach*: apply expression to model using interpreter, validate visually.
- 95 manual test cases – 4 logical errors, 1 user-friendliness issue, handful of code defects discovered and resolved.

Validation (cont’d)

Manual integration tests:

- *Strategy*: define basic input model, TDL^{TP} expression.
- *Approach*: apply expression to model using interpreter, validate visually.
- 95 manual test cases – 4 logical errors, 1 user-friendliness issue, handful of code defects discovered and resolved.



```
Parsing TDL expression.
TDL expression successfully parsed.
Parsing SUT model.
SUT model successfully parsed.
Composing scenario.
Scenario composition successful.
Expression reduced to:
E(TS2) & ~E(TS1)
Serializing scenario.
Scenario serialization successful.
```

Figure 13. Example integration test case.

Next Section

1. Context
2. Objective
3. Implementation
4. Validation
5. Summary

Summary

In summary:

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL$^{\text{TP}}$.

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL$^{\text{TP}}$.
- *Assessment*: successfully implemented, sufficiently validated.

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL^{TP}.
- *Assessment*: successfully implemented, sufficiently validated.
- The interpreter will find use in a prototype environment for MBT of cyber-physical systems under development at TalTech for several years;

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of $\mathrm{TDL}^{\mathrm{TP}}$.
- *Assessment*: successfully implemented, sufficiently validated.
- The interpreter will find use in a prototype environment for MBT of cyber-physical systems under development at TalTech for several years;
- Project management/code repository: TalTech GitLab.

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL^{TP}.
- *Assessment*: successfully implemented, sufficiently validated.
- The interpreter will find use in a prototype environment for MBT of cyber-physical systems under development at TalTech for several years;
- Project management/code repository: TalTech GitLab.

Future work:

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL^{TP}.
- *Assessment*: successfully implemented, sufficiently validated.
- The interpreter will find use in a prototype environment for MBT of cyber-physical systems under development at TalTech for several years;
- Project management/code repository: TalTech GitLab.

Future work:

- resolve major open questions in TDL^{TP} theory uncovered as part of this work (discussed in Sections 5.3, 5.2.2, Appendices 5 – 6 of the thesis);

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL^{TP}.
- *Assessment*: successfully implemented, sufficiently validated.
- The interpreter will find use in a prototype environment for MBT of cyber-physical systems under development at TalTech for several years;
- Project management/code repository: TalTech GitLab.

Future work:

- resolve major open questions in TDL^{TP} theory uncovered as part of this work (discussed in Sections 5.3, 5.2.2, Appendices 5 – 6 of the thesis);
- expand automated test coverage;

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL^{TP}.
- *Assessment*: successfully implemented, sufficiently validated.
- The interpreter will find use in a prototype environment for MBT of cyber-physical systems under development at TalTech for several years;
- Project management/code repository: TalTech GitLab.

Future work:

- resolve major open questions in TDL^{TP} theory uncovered as part of this work (discussed in Sections 5.3, 5.2.2, Appendices 5 – 6 of the thesis);
- expand automated test coverage;
- graphical user interface for annotating input model according to the TDL^{TP} input expression;

Summary

In summary:

- *Goal*: implement an interpreter based on the theory of TDL^{TP}.
- *Assessment*: successfully implemented, sufficiently validated.
- The interpreter will find use in a prototype environment for MBT of cyber-physical systems under development at TalTech for several years;
- Project management/code repository: TalTech GitLab.

Future work:

- resolve major open questions in TDL^{TP} theory uncovered as part of this work (discussed in Sections 5.3, 5.2.2, Appendices 5 – 6 of the thesis);
- expand automated test coverage;
- graphical user interface for annotating input model according to the TDL^{TP} input expression;
- codebase: apply generalizations where applicable.

Summary (cont'd)

toggl was used to record time spent on the thesis.

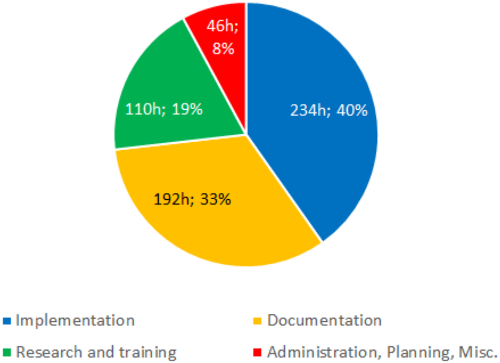


Figure 14. Overview of thesis scope.