computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

TEST SCENARIO SPECIFICATION LANGUAGE FOR MODEL-BASED TESTING

Evelin Halling [1], Jüri Vain [1], Artem Boyarchuk [2], Oleg Illiashenko [2]

[1] Tallinn University of Technology, Akadeemia tee 15A, Tallinn, Estonia
evelin.halling@taltech.ee, juri.vain@ttu.ee, http://www.taltech.ee
[2] Department of Computer Systems and Networks, National Aerospace University KhAI, Chkalova str., 17, 61070, Kharkov, Ukraine, boyarchuk@csn.khai.edu, o.illiashenko@khai.edu, http://csn.khai.edu

Paper history:
Received 08 April 2019
Received in revised form 10 October 2019
Accepted 02 December 2019
Available online 31 December 2019

Keywords:
Model-based testing;
Test scenario description language;
Timed automata;
Verification by model checking;
Conformance testing.

Abstract: In mission critical systems a single failure might cause catastrophic consequences. This sets high expectations to timely detection of design faults and runtime failures. By traditional software testing methods the detection of deeply nested faults that occur sporadically is almost impossible. The discovery of such bugs can be facilitated by generating well-targeted test cases where the test scenario is explicitly specified. On the other hand, the excess of implementation details in manually crafted test scripts makes it hard to understand and to interpret the test results. This paper defines high-level test scenario specification language TDL^{TP} for specifying complex test scenarios that are relevant for model-based testing of mission critical systems. The syntax and semantics of TDL^{TP} operators are defined and the transformation rules that map its declarative expressions to executable Uppaal Timed Automata test models are specified. The scalability of the method is demonstrated on the TUT100 satellite software integration testing case study.

1. INTRODUCTION

In model-based testing (MBT), the requirements model of System Under Test (SUT) describes the expected correct behavior of the system under possible inputs from its environment. The model, represented in a suitable machine interpretable formalism, can be used to automatically generate the test cases either offline or online, and be used as the oracle that checks if the SUT behavior conforms to this model. Offline test generation means that tests are generated before test execution and executed when needed. In online test generation the model is executed in lock step with the SUT. The test model communicates with SUT via controllable inputs and observable outputs of the SUT.

Test description in MBT typically relies on two formal representations, SUT modelling language and the test purpose specification language. An extensive survey on modelling formalisms used in MBT can be found in [20].

The requirements to the test purpose specification languages for MBT can be summarized as following:

1. intuitive and user-friendly specification process;
2. expressiveness to capture the features and behaviours under test in a compact and unambiguous form;
3. formal semantics to make the test purpose specifications verifiable and pertinent for automated test generation;
4. decidability to make the test generation from test purpose specification algorithmically feasible.

The first two criteria have been capitalized in earlier attempts of designing test purpose specification languages. Check Case Definition Language (CCDL) [1] provides a high-level approach for requirements-based black-box system level testing. Test simulations and expected results specified in human readable form in CCDL can be compiled into executable test scripts. However, due to the lack of standardization, high-level tests in CCDL are heavily tool-dependent and can be used only in tool specific testing processes.

High-level keyword-based test languages, such as the Robot Framework [2], have also been integrated with MBT [3]. In domains such as avionics [4] and automotive industry the efforts have been made to address the standardization of testing methods and languages, e.g. creating a meta-model for testing avionics systems [4], and the Automotive TestML [5]. Similarly, the Open Test Sequence Exchange Format (OTX) [6] standardized by ISO provides tool-independent XML-based data exchange format [7] for description and documentation of executable test sequences. These efforts have focused primarily on enabling the exchange of test specifications between involved stakeholders and tools. Due to their domain and purpose specialization the applicability of these languages in other domains is limited.

The Message Sequence Chart (MSC) [8] standardized by International Telecommunication Union was one of the first scenario specification languages though it was not only focusing on testing. The semantics of MSC is specified in [9]. Some of the features of MSC are adopted in UML, e.g. in Sequence Diagrams. Still, loose semantics limits its use as a consistent test description language [10].

Precise UML [11] introduces a subset of UML and OCL for MBT. The attempt to unify the semantics of different diagrams was motivated by the need for behavioral specifications of SUT which are well suited for generating test cases out of SUT models.

Concrete test scripting languages, such as TTCN-3, regardless their strict semantics are not well suited for high-level description of test scenarios. They rather follow the style of syntax typical to imperative programming languages [12].

Thus, most of the test purpose specification languages referred above suffer from some of the disadvantages, either they have imprecise or informal semantics, lack of standardization, lack of comprehensive tool support, or poor interoperability with other development and testing tools.

European Telecommunications Standards Institute (ETSI) intended to address these shortcomings and developed a new specification language standard by introducing Test Purpose Language (TPLan) that supports the high-level expression of test purposes in prose [13]. Though TPLan provides notation for the standardized specification of test purposes, it leaves a gap between the declarative test purpose and its imperative implementation in test. Without formal semantics the development of test descriptions by means of different notations and dialects led to overhead and inconsistencies that need to be checked and fixed manually. As a consequence, ETSI started a new initiative by developing the Test Description Language TDL [12]. It is intended to bridge the gap between declarative test purposes and imperative test cases by offering a standardized approach for the specification of test descriptions. The main benefits of ETSI TDL outlined in [12] are higher quality tests through better design, easier layout to review by non-testing experts, better and faster test development, and seamless integration of methodology and tools.

The development of ETSI TDL was driven by industry where it is used primarily, but not exclusively, for functional testing. To enable the application of TDL in UML based working environments, a UML Profile for TDL (UP4TDL) [10] was developed. Domain-specific concepts are represented in UP4TDL by means of stereotypes.

Though TDL features one of the most advanced test purpose description language it has room for improvements. In the first place, automatic mapping of ETSI TDL to TTCN-3 is not fully implemented yet. The mapping is needed for generating executable tests from TDL descriptions and re-using the existing TTCN-3 tools and frameworks for test execution.

Second limitation of TDL is restricted timing semantics. The Time package in TDL contains concepts for the specification of time operations, time constraints, and timers. Since time in TDL is global and progresses monotonically in discrete quantities there is no way of expressing synchronization conditions between local time events of parallel processes and detecting possible Zeno computations that can be analyzed in continuous time models. Similarly, timelock-freedom and bifurcation analysis [22] cannot be performed in this setting.

One step further towards automatic test generation was timed games based synthesis of test strategies introduced in [14] and implemented in the Uppaal Tiga tool. Timed Computation Tree Logic (TCTL) used for specifying test purpose in this approach has high expressive power and formal semantics relevant for expressing quantitative time properties combined with CTL operators such as *'always'*, *'inevitable'*, *'potentially always'*, '*possible*', and *'leads-to'* [15].

Due to the complexity of model checking [21], the TCTL syntax in Uppaal tool is limited with un-nested operators making the TCTL expressions flat with respect to the temporal operators. On the other hand, to specify the properties of timed reachability the flat TCTL expressions are not sufficient for specifying complex properties and so called auxiliary property recognizing automata, e.g. 'stopwatch' automata are needed. Modifying the test model structure by adding property automata is not

trivial for non-experts and may be an error prone process leading to the unintended changes of semantics of tests.

The aim of this work is to build an extra language layer (Test Scenario Definition Language - TDL^{TP}) for test scenario specification that is expressive, free from the limitations of 'flat' TCTL, interpretable in Uppaal TA, and suited for test generation.

In our approach, Uppaal Timed Automata (TA) [16] serve as a SUT specification language. Uppaal TA have been chosen because they are designed to express the timed behavior of state transition systems and there exists a mature set of tools that supports model construction, verification and online model-based testing [17].

For the test purpose specification to be concise and still expressive its specification language must be more abstract than SUT modeling language and not necessarily self-contained in the sense that its expressions are interpreted in the context of SUT model only. It means that the terms of test purpose specification refer to the SUT model structural elements of interest, they are called test coverage items (TCIs). The test purpose specification language TDL^{TP} proposed in our approach allows expressing multiple coverage criteria in terms of TCIs, including test scenario constraints such as *iteration*, *next*, *leads to*, and structural coverage criteria such as *selected states*, *selected transitions*, *transition pairs*, and timing constraints, e.g. *time bounded leads to*.

Generating the test model based on the SUT model and TDL^{TP} coverage expression includes two phases.

In the first phase, the TCIs have to be labelled in the SUT model with Boolean variables called *traps*. The traps are needed to make TCIs referable in the TDL^{TP} expressions. In case of non-deterministic SUT model the coverage of those elementary TCIs is ensured by reactive planning tester (RPT) automata, one automaton for each conjunctive set of TCIs (see [19] for further details of RPT generation).

In the second phase of generation, a test supervisor model M^{SVR} is constructed from the TDL^{TP} expression to trigger the RPT automata according to the test scenario so that the temporal and logical coverage constraints stated in TDL^{TP} specification would be satisfied. Since non-deterministic SUT models based tests are partially controllable only pseudo optimal traces can be generated by this method.

Alternatively, in case of deterministic SUT models, the RPT automata generation phase can be discarded since Uppaal model checker generates optimal witness traces from the parallel composition of SUT and tester models.

The rest of this paper is organized as follows. In Section 2 Uppaal Timed Automata formalism is introduced, Sections 3 and 4 define the TDL^{TP} language syntax and semantics respectively, Section 5 defines the map from TDL^{TP} to Uppaal TA that controls if the test scenario execution satisfies its declarative expression. In Section 6 the reduction rules of TDL^{TP} expressions are presented. Section 7 describes how the whole test model is composed by introducing test supervisor automaton. Section 8 explains how the test verdict and test diagnosis capability are encoded in the tester model, and finally the conclusions are drawn.

2. UPPAAL TIMED AUTOMATA

Uppaal Timed Automata [16] (TA) used for modelling SUT is defined as a closed network of extended timed automata that are called *processes*. The processes are gathered into a single system by parallel composition known from the process algebra CCS. An example of a system comprising two automata is given in Fig. 1.

The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by double circle inside the location.

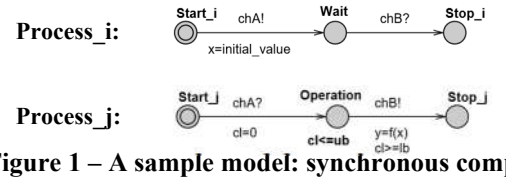


Figure 1 – A sample model: synchronous composition of two Uppaal automata Process_i and Process_j

Synchronous communication between the processes is by hand-shake synchronization links that are called *channels*. A channel relates a pair of edges labeled with symbols for input actions denoted by e.g. chA? and chB? in Fig. 1, and output actions denoted by chA! and chB!, where chA and chB are the names of the channels.

In Fig. 1, there is an example of a model that represents a synchronous remote procedure call. The calling process Process_i and the callee process Process_j both include three locations and two synchronized transitions. Process_i, initially at location Start_i, initiates the call by executing the send action chA! that is synchronized with the receive action chA? in Process_j. The location Operation denotes the situation where Process_j computes the value to output variable y. Once done, the control is returned to Process_i by the action chB!.

The duration of executing the result is specified by the interval [*lb, ub*] where the upper bound *ub* is

given by the *invariant* cl<=ub of location Operation, and the lower bound *lb* by the guard condition cl>=lb of the transition Operation ⟶ Stop_j. The *assignment* cl=0 on the transition Start_j ⟶ Operation ensures that the clock cl is reset when the control reaches the location Operation. The global variables x and y model the input and output arguments of the remote procedure call, and the computation itself is modelled by the function f(x) defined in the declarations block.

While the synchronous communication between processes is modeled using channels, asynchronous communication between processes is modeled using global variables accessible to all processes.

Formally, the Uppaal TA are defined as follows:

Let $\sum$ denote a finite alphabet of actions *a, b, ...* and *C* a finite set of real-valued variables *p, q, r*, denoting clocks. A guard is a conjunctive formula of atomic constraints of the form $p \sim n$ for $p \in C$, $\sim \in \{\leq, \geq, =, <, >\}$ and $n \in N^+$. We use *G(C)* to denote the set of clock guards. A timed automaton *A* is a tuple $\langle N, l_0, E, I\rangle$ where *N* is a finite set of locations (graphically denoted by nodes), $l_0 \in N$ is the initial location, $E \in N \times G(C) \times \sum \times 2^C \times N$ is the set of edges (an edge is denoted by an arc) and $I : N \longrightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $p \leq n$ or $p < n$, $n \in N^+$.

Without the loss of generality we assume that guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer variables. Similarly to clock conditions, the propositions on integer variables *k* are of the form $k \sim n$ for $n \in N$, and $\sim \in \{\leq, \geq, =, <, >\}$. For the formal definition of Uppaal TA semantics we refer the reader to [18] and [16].

3. TDL^{TP} SYNTAX

The ground terms in TDL^{TP} are sets (denoted *TS*) of assignments to auxiliary variables called *trap variables* or simply *traps* added to the SUT model for test purpose specification. A trap is updated by Boolean variable assignment that labels a TCI. In case of Uppaal TA, the TCIs are edges of the SUT model M^{SUT}. The value of all traps is initially set to *false*. When the edge of M^{SUT} labelled with a trap is visited during test execution the trap update function is executed and the trap value is set to *true*. We say that a trap *tr* is *elementary trap* if its update function is unconditional, i.e. of shape *tr* := *true*.

Generally we assume that the trap names are unique, trap update functions are non-recursive and their arguments have definite values whenever the edge labelled with that trap is executed. The trap *tr* update condition, if *conditional trap*, is a Boolean expression (we call it also update constraint) the arguments of which range over the sets of variables and constants of M^{SUT} and over the auxiliary constants and variables occurring in the test purpose specification in TDL^{TP}, e.g. references to other traps, event counters and the time bounds of model clocks.

Although we deal with finite sets of traps and their value domains the quantifiers are introduced in TDL^{TP} for notational convenience. To refer to the situations where many traps have to be true or false at once, we group these traps to sets called *trapsets* denoted by *TS* and prefix them with trapset quantifiers - A for universal and E for existential quantification. *A*(*TS*) means that all traps and *E*(*TS*) means that at least one trap of the set *TS* has to be true. To represent a trapset in Uppaal TA syntax we encode them as one-dimensional trap arrays and refer to individual traps in the array by array index value, e.g. *i*-th trap in *TS* is referred to as *TS*[*i*].

In the following we give the syntax of TDL^{TP} expressions in BNF:

⟨*Expression*⟩ ::=
'(' ⟨*Expression*⟩ ')'
| *'A'* ⟨*TrapsetExpression*⟩
| *'E'* ⟨*TrapsetExpression*⟩
| ⟨*UnaryOp*⟩ ⟨*Expression*⟩
| ⟨*Expression*⟩ ⟨*BinaryOp*⟩ ⟨*Expression*⟩
| ⟨*Expression*⟩ ~> ⟨*Expression*⟩
| ⟨*Expression*⟩ ~>'['⟨*RelOp*⟩⟨*NUM*⟩']' ⟨*Expression*⟩
| *'#'* ⟨*Expression*⟩ ⟨*RelOp*⟩ ⟨*NUM*⟩

⟨*TrapsetExpression*⟩ ::=
'(' ⟨ *TrapsetExpression*⟩')'
| '!' ⟨*ID*⟩
| ⟨*ID*⟩ *'\'* ⟨*ID*⟩
| ⟨*ID*⟩ *';'* ⟨*ID*⟩

⟨*UnaryOp*⟩ ::= *'not'*
⟨*BinaryOp*⟩ ::= *'&'* | *'or'* | *'=>'* | *'<=>'*
⟨*RelOp*⟩ ::= *'<'* | *'='* | *'>'* | *'<='* | *'>='*
⟨*ID*⟩ ::= (*'TR'*) ⟨*NUM*⟩
⟨*NUM*⟩ ::= ('0'..'9')+

4. TDL^{TP} SEMANTICS

To define the semantics of *TDL^{TP}* we assume there are given:

- an Uppaal TA model *M*;
- *Trapset TS* which is possibly a union of member trapsets $TS = \bigcup_{i=1,m} TS_i$, where the cardinality of each TS_i is n_i;
- $L: TS \longrightarrow E(M)$, the labelling function that maps the traps in *TS* to edges in *E*(*M*), where *E*(*M*) denotes the set of edges of the model *M*. We assume the *uniqueness* of the labeling within a trapset, i.e. there is at most one edge

labelled with a trap from the given trapset but an edge can be labelled with many traps if each of them is from different trapset.

4.1 ATOMIC LABELLING FUNCTION

Atomic labelling function is non-surjective and injective-only mapping between *TS* and $E(M)$, i.e. each element of *TS* is mapped to a unique edge in $E(M)$:

$$\begin{gathered}\mathrm{L}\colon \mathrm{TS} \longrightarrow \mathrm{E(M)},\ \text{s.t.}\ \forall \mathrm{e} \in \mathrm{E(M)}: \\ \mathrm{TS_k[i]} \in \mathrm{L(e)} \wedge \mathrm{TS_l[j]} \in \mathrm{L(e)} \Rightarrow \mathrm{k} \neq \mathrm{l}.\end{gathered} \quad (1)$$

4.2 DERIVED LABELLING OPERATIONS (TRAPSET OPERATIONS)

The formulas with a trapset operation symbol and trapset(s) identifiers being its argument(s) are called TDL[TP] trapset formulas.

***Relative complement of trapsets* ($TS_1 \backslash TS_2$).** Only those edges labelled with traps of TS_1 and not with traps of TS_2 are in the relative complement trapset $TS_1 \backslash TS_2$:

$$\begin{gathered}[\![TS_1 \backslash TS_2]\!] \ \textit{iff} \\ \forall i \in [0, n_1], j \in [0, n_2], \exists e \in E(M): \\ TS_1[i] \in L(e) \wedge TS_2[j] \notin L(e).\end{gathered} \quad (2)$$

***Absolute complement of a trapset* (!TS).** All edges that are not labelled with traps of TS are in the absolute complement trapset !TS:

$$[\![!\,TS]\!] \ \textit{iff} \ \forall i \in [0, n], \exists e \in E(M): TS[i] \notin L(e). \quad (3)$$

***Linked pairs of trapsets* (TS_1; TS_2).** Two trapsets TS_1 *and* TS_2 are linked via operator *next* (denoted ';') if and only if there exists a pair of edges in M which are labelled with traps of TS_1 and TS_2 respectively and which are connected through a location so that if any of traps in TS_1 is updated to true on the k-th transition of model M execution trace σ then some trap of TS_2 is updated to true in the (k+1)-th transition of that trace:

$$\begin{gathered}[\![TS_1;\ TS_2]\!] \quad \textit{iff} \quad \forall i \in [0, n_1], \exists j \in \\ [0, n_2], \sigma, k\colon \ [\![TS_1[i]]\!]_{\sigma^k} \Rightarrow [\![TS_2[j]]\!]_{\sigma^{k+1}},\end{gathered} \quad (4)$$

where $[\![TS]\!]_\sigma$ denotes the interpretation of the trapset TS on the trace σ and σ^l denotes the *l*-th suffix of the trace σ, i.e. the suffix which starts from *l*-th location of σ; n_1 and n_2 denote cardinalities of trapsets TS_1 and TS_2 respectively. Note that operator ';' enables expressing one of the "classical" structural coverage criteria 'selected transition pairs'.

4.3 INTERPRETATION OF TDL EXPRESSIONS

***Quantifiers of trapsets*.** Given the definitions 1 - 4 of trapset operations we define the semantics of bounded universal quantifier *A* and bounded existential quantifier *E* of a trapset *TS* as follows:

$$[\![A\ (TS)]\!] \ \textit{iff} \ \forall i \in [0, n]: TS[i], \quad (5)$$
$$[\![E\ (TS)]\!] \ \textit{iff} \ \exists i \in [0, n]: TS[i], \quad (6)$$

where *n* denotes the cardinality of the trapset *TS*.

Note that quantification is defined on the trapsets only and not on higher level operators.

***Logic connectives*.** Since recursive nesting of TDL[TP] logic and temporal operators is allowed for better expressiveness we define the semantics of these higher level operators where the argument terms are not trapset formulas but derived from them using recursive nesting of logic and temporal operator symbols. Let *SE*, SE_1 and SE_2 denote such argument sub-formulas, then

$$[\![SE_1\ \&\ SE_2\ ]\!] \quad \textit{iff} \quad [\![SE_1]\!]\ \textit{and}\ [\![SE_2]\!] \quad (7)$$
$$[\![SE_1\ \textit{or}\ SE_2\ ]\!] \quad \textit{iff} \quad [\![SE_1]\!]\ \textit{or}\ [\![SE_2]\!] \quad (8)$$

$$SE_1 \Rightarrow SE_2 \equiv not(SE_1) \vee SE_2 \quad (9)$$

$$\begin{gathered}SE_1 <\!\!=\!\!> SE_2 \equiv (SE_1 \Longrightarrow SE_2) \wedge \\ (SE_2 \Longrightarrow SE_1).\end{gathered} \quad (10)$$

Temporal operators

Leads to' operator $'SE_1 \rightsquigarrow SE_2'$ in TDL[TP] is inspired by Computation Tree Logic CTL 'always leads to' operator, denoted by $'\varphi - -> \psi'$ in Uppaal, which is equivalent to CTL formula $A\Box(\varphi \Longrightarrow A\Diamond\psi)$. Leads to expresses that after reaching the state which satisfies φ in the computation all possible continuations of this computation reach the state in which ψ is satisfied. For clarity we substitute the meta-symbols φ and ψ with non-terminals SE_1 and SE_2 of TDL[TP].

$$\begin{gathered}[\![SE_1 \sim > \ SE_2\ ]\!] \ \ \textit{iff} \\ \forall \sigma, \exists k, l, k \leq l\colon [\![SE_1]\!]_{\sigma^k} \Longrightarrow [\![SE_2]\!]_{\sigma^l},\end{gathered} \quad (11)$$

where σ^k denotes the *k*-th suffix of the trace σ, i.e. the suffix which starts from *k*-th location of σ, and $[\![SE]\!]_{\sigma^k}$ denotes the interpretation of *TS* on the *k*-th suffix of trace σ.

'Time bounded leads to' means that TS_2 must occur after TS_1 and the time instance of TS_2 occurrence (measured relative to TS_1 occurrence satisfies constraint $\circledast\, n$, where $\circledast \in \{<, =, >, \leq, \geq\}$ and $n \in \mathbf{N}$:

$$[\![SE_1 \sim >_{[\circledast n]}\ SE_2]\!] \quad iff \quad \forall\sigma, \exists k, l, k \le l: [\![SE_1]\!]_{\sigma^k} \Longrightarrow [\![SE_2]\!]_{\sigma^l}. \qquad (12)$$

'Conditional repetition'. Let k enumerate the occurrences of $[\![SE]\!]$, then

$$[\![\#SE \circledast n]\!] \ iff \rightsquigarrow \cdots \rightsquigarrow [\![SE]\!]^k \ and \ k \circledast n, \qquad (13)$$

where index variable k satisfies constraint $\circledast n$, $\circledast \in \{<, =, >, \le, \ge\}$ and $n \in \mathbf{N}$.

The application of logic *not* to non-ground level TDL^{TP} terms has following interpretation:

$$not\big(A\ (TS)\big)\ iff \quad \exists i: [\![TS[i]]\!] = false \qquad (14)$$

$$not\big(E\ (TS)\big)\ iff \quad \forall i: [\![TS[i]]\!] = false \qquad (15)$$

$$not\ (SE_1 \wedge SE_2) \equiv not\ (SE_1)\ \vee\ not\ (SE_2) \qquad (16)$$

$$not\ (SE_1 \vee SE_2) \equiv not\ (SE_1)\ \wedge\ not\ (SE_2) \qquad (17)$$

$$not\ (SE_1 \Longrightarrow SE_2) \equiv SE_1\ \wedge\ not\ (SE_2) \qquad (18)$$

$$not\ (SE_1 \Longleftrightarrow SE_2 \equiv not\ (SE_1 \Longrightarrow SE_2)\ \vee\ not\ (SE_2 \Longrightarrow SE_1) \qquad (19)$$

$$[\![not(SE_1 \rightsquigarrow\ SE_2)]\!]\ iff\ [\![not\ (SE_1)]\!]\ or\ \forall k, l, k \le l: [\![SE_1]\!]_{\sigma^k}\ and\ not[\![SE_2]\!]_{\sigma^l} \qquad (20)$$

$$not\ (SE_1 \rightsquigarrow_{\circledast n}\ SE_2) \equiv not\big(SE_1 \rightsquigarrow SE_2\big) \vee \forall\phi: (SE_1 \rightsquigarrow_{\phi}\ SE_2\ ) \Rightarrow (\phi \Rightarrow not(\circledast n)), \qquad (21)$$

$$not\ (\#TS\ \circledast n) \equiv \forall\phi: (\#TS\ \phi) \Rightarrow (\phi \Rightarrow not(\circledast n)\ ), \qquad (22)$$

where ϕ denotes the time bound constraint that yields the negation of constraint $\circledast n$.

5. MAPPING *TDL^{TP}* EXPRESSIONS TO BEHAVIOR RECOGNIZING AUTOMATA

When mapping the TDL^{TP} formulae to test supervisor component automata we implement the mappings starting from ground level terms and move towards the root term by following the structure of the TDL^{TP} formula parse tree. The terminal nodes of any TDL^{TP} formula parse tree are trapset identifiers. The next above the terminal layer of the parse tree constitute the trapset operation symbols. The trapset operation symbols, in turn, are the arguments of logic and temporal operators. The ground level trapsets and the trapsets which are the results of trapset operations are mapped to the labelling of SUT model M^{SUT}. In the following the mappings are specified for TDL^{TP} trapset operations, logic operators and temporal operators in separate subsections.

5.1 MAPPING *TDL^{TP}* TRAPSET EXPRESSIONS TO SUT MODEL M^{SUT} LABELLING

*Mapping M*1: *Relative complement of trapsets* $TS_1 \backslash TS_2$: The $TS_1 \backslash TS_2$ – mapping adds the traps of the trapset $TS_1 \backslash TS_2$ only to these edges of M^{SUT} which are labelled with traps of TS_1 and not with traps of TS_2. An example of such mapping is depicted in Fig. 2.

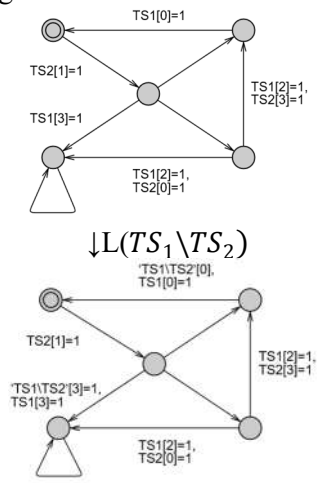


Figure 2 – Mapping TDL^{TP} expression $TS_1 \backslash TS_2$ to the SUT model labelling

*Mapping M*2: *Absolute complement of a trapset* $!TS$: The mapping of $!TS$ to SUT model labelling provides the labelling with $!TS$ traps all such edges of SUT model M^{SUT} which are not labelled with traps of *TS*. Example of this mapping is depicted in Fig. 3.

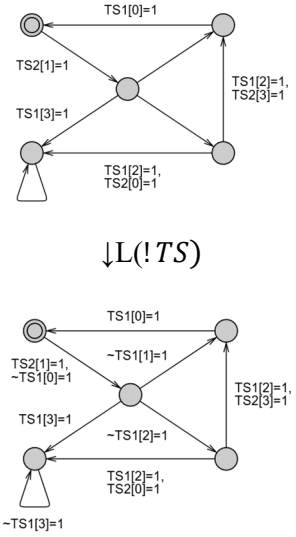


Figure 3 – Mapping TDL^{TP} expression $!TS$ to the SUT model labelling

*Mapping M*3: *Linked pairs of trapsets* $TS_1; TS_2$:

The mapping of terms $TS_1; TS_2$ to labelling is implemented by the labelling algorithm *Algorithm* 1 ($L(TS_1; TS_2)$)

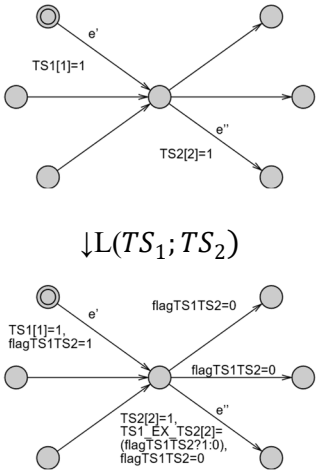


Figure 4 – Example of the application of ALGORITHM 1 ($L(TS_1; TS_2)$)

The example of Algorithm 1 application is demonstrated in Fig. 4. Notice that the labelling concerns not only the edges that are labelled with traps of *TS1* and *TS2* but also those which depart from the same location as the edge with *TS2* labelling. This is necessary for resetting the variable *flag* which indicates the executing a trapset *TS1* labelled edge in the previous step of the computation.

$$
\begin{aligned}
&\boldsymbol{forall}\\
&\quad e', e'', i, j: pre(e'') = post(e') \wedge TS_1[i] \in L(e')\\
&\quad \boldsymbol{if}\ TS_2[j] \in L(e'')\\
&\quad\quad \boldsymbol{then}\\
&\quad\quad\quad Asg(e') \leftarrow Asg(e'), flag(TS_1; TS_2)\\
&\quad\quad\quad\quad = true,\\
&\quad\quad\quad Asg(e'') \leftarrow Asg(e''), TS(TS1; TS2)[j]\\
&\quad\quad\quad = (flag(TS_1; TS_2)?\, true: false),\\
&\quad \boldsymbol{fi}\\
&\quad Asg(e'') \leftarrow Asg(e''), flag(TS_1; TS_2) = false\\
&\boldsymbol{end\ forall}
\end{aligned}
$$

5.2 MAPPING TDL^{TP} LOGIC OPERATORS TO RECOGNIZING AUTOMATA

The indexing of trapset array elements, universal and existential quantifiers in Uppaal modelling language support direct mapping of trapset quantifiers to *forall* and *exists* expressions of Uppaal TA as shown in Fig. 5 and 6.

*Mapping M*4: *Universal quantifier of the trapset*

Figure 5 – An automaton that recognizes universally quantified trapset expressions

*Mapping M*5: *Existential quantifier of the trapset*

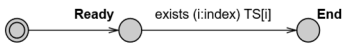


Figure 6 – The automaton that recognizes existentially quantified trapset expressions

Negation not

Since logic negation *not* can be pushed to ground level trapset terms by applying equivalences (14 – 22) and the direct mappings of *not* formulas are not considered in this work.

*Mapping M*6: *Conjunction of sub-formulas*

The conjunction *SE*1 & *SE*2 is mapped to the automata fragment as shown in Fig. 7. In the conjunction and disjunction automata depicted in the Fig. 7 and 8 the guard conditions *P* and *Q* encode the argument terms *SE*1 and *SE*2 respectively. In conjunction automaton the *End* location is reachable from the initial location *Idle* if both *P* and *Q* evaluate to true in any order.

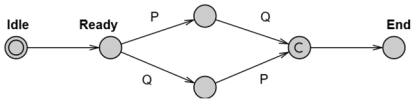


Figure 7 – The automaton that recognizes the conjunction of TDL^{TP} formulas P and Q

*Mapping M*7: *Disjunction of sub-formulas*

In the disjunction automaton the *End* location is reachable from the initial location *Idle* if either *P* and *Q* are true.

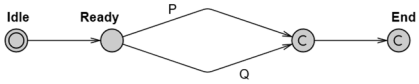


Figure 8 – Automaton that recognizes the disjunction of TDL^{TP} formulas P and Q

The *implication* of *TDL^{TP}* formulas can be defined using disjunction and negation as shown in formula (9) and their transformation to property automata are implemented through these mappings.

Similarly, the *equivalence* of TDL^{TP} formulas can be expressed via conjunction and implication by using equivalence in formula (10).

5.3 MAPPING TDL^{TP} TEMPORAL OPERATORS TO RECOGNIZING AUTOMATA

*Mapping M*8: *'Leads to' $p \rightsquigarrow q$*

Mapping the *leads to* operator to Uppaal TA produces the model fragment depicted in Fig. 9.

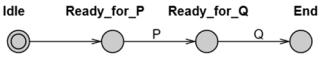
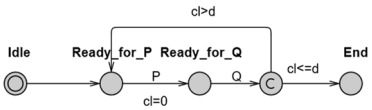


Figure 9 – 'Leads to' formula $p \rightsquigarrow q$ recognizing automaton

Mapping M9: Timed leads to $p \rightsquigarrow_{\circledast con}\ q$

Mapping *'timed leads to'* to a Uppaal TA fragment is depicted in Fig. 10. It presumes an additional clock cl which is reset to 0 at the time instant when formula *P* become true. The condition 'cl<=d' in Fig. 10 a) sets the upper time bound d to the event when formula *Q* has to becomes true after *P*, i.e. after the clock cl reset.

a)



b)

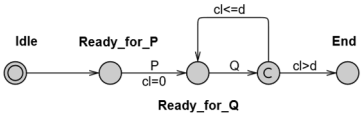


Figure 10 – 'Timed leads to' formula P $\rightsquigarrow_{\circledast d}$ *Q* recognizing automata a) with condition cl≤d; b) with condition cl>d

The mapping to property automaton depends on the time condition of *leads to*. For instance if the conditions is 'cl>d' the mapping results in automaton shown in Fig. 10 b).

*Mapping M*10: *Conditional repetition* $\#SE \circledast n$:

The Uppaal TA fragment generated by the mapping of $\#SE \circledast n$ (Fig. 11) includes a counter variable *i* to enumerate the events when the *SE* formula *P* becomes *true*. If the loop exit condition, e.g., 'i >=n', is satisfied then the transition to location *End* is fired without delay (the middle location is of type *committed*).

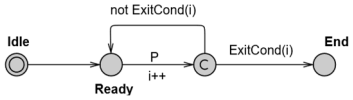


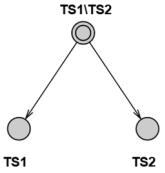
Figure 11 – Uppaal TA that implements conditional repetition

6. REDUCTION OF THE SUPERVISOR AUTOMATA AND THE LABELLING OF SUT

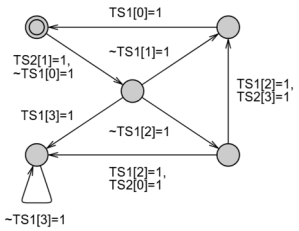
The TDL^{TP} expressions with many nested operators may become large and involve some overhead. Removal of this overhead in the formulas provides reduction in the state space needed for their model checking and improves the readability and comprehension of this formula.

The simplifications are formulated in terms of the parse tree of the TDL^{TP} formula and standard logic simplifications. Due to the nesting of operations in the TDL^{TP} formula the root operation can be any operator listed in the BNF grammar of TDL^{TP} but the terminals of the parse tree are always trapsets.

a)



b)



c)

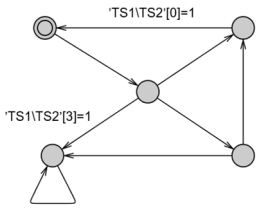


Figure 12 – Simplification of $TS_1 \setminus TS_2$ trapsets labelling: a) the parse tree of $TS_1 \setminus TS_2$; b) labelling of the SUT model with TS_1, TS_2 and $TS_1 \setminus TS_2$ c) reduced labelling of the SUT model M^{SUT}

TDL^{TP} formulas consist of a static component (a trapset or a trapset expression) and optionally the logic and/or temporal component. The static component includes all sub-formulas of the parse tree branches from terminals to the lowest temporal expression, all sub-formulas above it are temporal and/or logic formulas (possibly mixed).

The trapset formulas are implemented by labelling operations such as *relative* and *absolute complement*. Only trapset formulas can be universally and existentially quantification. No nesting of quantifiers is allowed. Since the validity of root formula can be calculated only using the truth value of the highest trapset expression in the parse tree, all the trapsets being closer to the ground level trapset along the parse tree sub branch can be removed from the labelling of the SUT model. This reduction can be done after labelling the SUT model and applying all the trapset operations. An example of such reduction is demonstrated for relative complement operation $TS_1 \setminus TS_2$ in Fig. 12.

Logic simplification follows after the trapset expression simplification is completed. Here standard logic simplifications are applicable:

$$\begin{aligned} &p \wedge p \equiv p \\ &p \wedge not\ p \equiv false, \\ &p \wedge false \equiv false, \\ &p \wedge true \equiv p, \\ &p \vee p \equiv p, \\ &p \vee not\ p \equiv true, \\ &p \vee false \equiv p, \\ &p \vee true \equiv true, \end{aligned} \tag{23}$$

We will introduce also a set of simplifications for TDL^{TP} temporal operators which follow from their semantics and the properties of integer arithmetic:

$$
\begin{aligned}
&TS \equiv false \; if \; TS = \emptyset \\
&p \rightsquigarrow false \equiv false \\
&false \rightsquigarrow p \equiv false \\
&true \rightsquigarrow p \equiv p \\
&p \rightsquigarrow true \equiv true \\
&\#p = 1 \equiv p \\
&\#p \circledast n_1 \wedge \#p \circledast n_2 \\
&\qquad \equiv \#p \circledast max(n_1, n_2) \;\; if \\
&\qquad \circledast \in \{\geq, >\} \\
&\#p \circledast n_1 \vee \#p \circledast n_2 \\
&\qquad \equiv \#p \circledast min(n_1, n_2) \;\; if \;\; \circledast \\
&\qquad \in \{\geq, >, =\} \\
&\#p \circledast n_1 \wedge \#p \circledast n_2 \equiv false \; if \;\; \circledast \\
&\qquad \in \{=\} \; and \; n_1 \neq n_2 \\
&\#p \circledast n_1 \;\rightsquigarrow\; \#p \circledast n_2 \\
&\qquad \equiv \#p \circledast (n_1 + n_2) \quad if \;\; \circledast \\
&\qquad \in \{\geq, >, =\} \\
&\#p \circledast n_1 \;\rightsquigarrow\; \#p \circledast n_2 \\
&\qquad \equiv \#p \circledast min(n_1, n_2) \;\; if \;\; \circledast \\
&\qquad \in \{<\} \\
&\#p \circledast n_1 \;\wedge \#p \circledast n_2 \\
&\qquad \equiv \#p \circledast min(n_1, n_2) \;\; if \;\; \circledast \\
&\qquad \in \{<\} \\
&\#p \circledast n_1 \;\vee \#p \circledast n_2 \\
&\qquad \equiv \#p \circledast max(n_1, n_2) \; if \;\; \circledast \\
&\qquad \in \{<\} \\
&p \rightsquigarrow_{d_1} q \;\wedge\; p \rightsquigarrow_{d_2} q \equiv \\
&p \rightsquigarrow_{\min(d_1, d_2)} q \quad if \; \circledast \in \{\leq, <\} \\
&p \rightsquigarrow_{d_1} q \;\wedge\; p \rightsquigarrow_{d_2} q \\
&\qquad \equiv p \rightsquigarrow_{\max(d_1, d_2)} q \quad if \; \circledast \\
&\qquad \in \{>\}
\end{aligned}
\tag{24}
$$

7. COMPOSING THE TEST SUPERVISOR MODEL

The test supervisor model M^{SVR} is constructed as a parallel composition of single TDL^{TP} property recognizing automata each of which is produced by parsing the TDL^{TP} formula and mapping corresponding sub-formulae to the automaton template as defined in Section 5. To interrelate these sub-formula automata, two phases have to be completed:

1) Each trap labelled transition *e* of M^{SUT} (here we consider the traps which are left after labels reduction as described in Section 6) has to be split in two edges *e*' and *e*" connected via an auxiliary committed location l^c. The edge *e*' will inherit the labelling of *e* while *e*" will be labelled with an auxiliary broadcast channel that signals the trap update occurrence to the upper neighbor sub-formula automaton. We use the channel naming convention, where a channel name has a prefix *ch_* followed by the trapset identifier, e.g. for an edge *e* labelled with the trap *TS*[*i*], the broadcast channel label *ch_TS*! is added to the edge *e*" (an example is shown in Fig. 13 a)).
2) Each non-trapset formula automaton will be extended with a wrapping construct shown in Fig. 13 b). The wrapper has one or two, channel labels, depending if the sub-formula operation is unary or binary, to synchronize its state transition with those of its child expression(s). We call them downwards channels denoted by *Activate_subOP1*, *Activate_subOP2* and used to activate the recognizing mode in the first and second sub-formula automata. Similarly, two broadcast channels are introduced to synchronize the state transition of sub-formula automata with their upper operation automaton. We call them upwards channels, denoted by *Activate_OPi* and *Done OPi* in Fig. 13 b). The root node is an exception because it has upwards channel only with the test *Stopwatch* automaton (the *Stopwatch* automaton will be explained in Section 8). If the sub-formulas of given property automaton are mapped to trapset expressions then the back edge *End*→*Idle* to the initial state is labelled also with trapset reset function with *TS* being the argument trapset identifier. The TDL^{TP} operator automata extensions with wrapper constructs for implementing their composition in test supervisor model M^{SVR} are shown in Fig. 14.

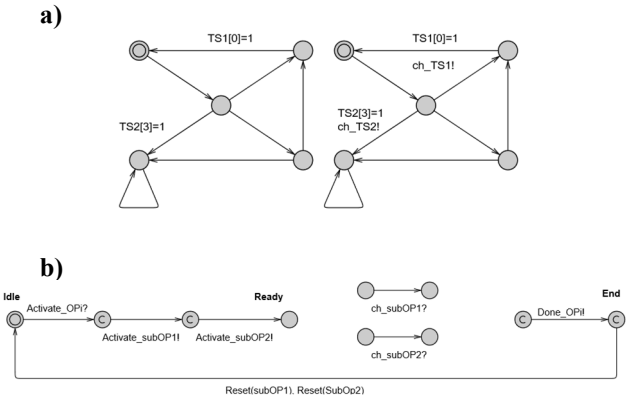


Figure 13 – a) Extending the trap labelled edges with synchronization conditions for composing the test supervisor; b) the wrapper pattern for composing operation recognizing automata

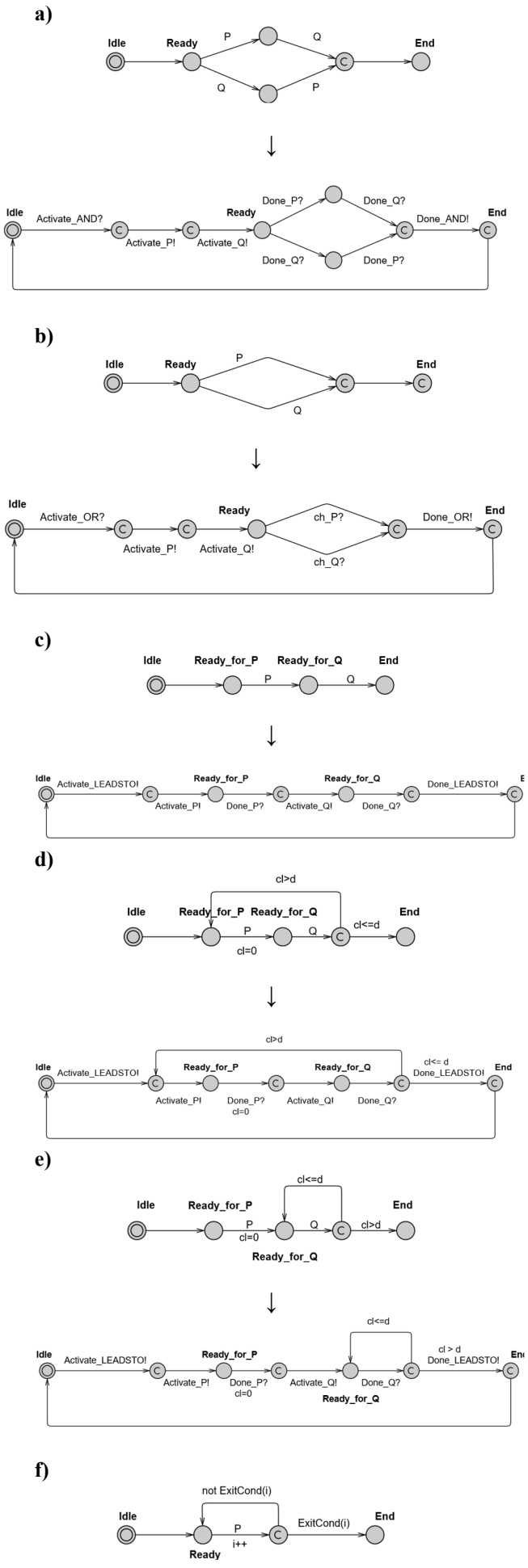


Figure 14 – Extending sub-formula automata templates with wrapping for test Supervisor composition a) And; b) Or; c) Leads to; d) Timed leads to with condition cl≤d; e) Timed leads to with condition cl>d; f) Conditional repetition

Note that the TDL^{TP} sub-formula meta symbols *P* and *Q* in the original templates are replaced with channels which signal when the sub-formulas interpretation automata reach their local *End* locations.

8. ENCODING THE TEST VERDICT AND TEST DIAGNOSTICS IN THE TESTER MODEL

The test verdict is yielded by the test *StopWatch* automaton either when the automaton reaches its end state *End* within time bound *TO*. Otherwise, the *timeout* event *Swatch==TO* triggers the transition to the terminal location *Failed*. Specifically, *Passed* in the *StopWatch* automaton is reached simultaneously with executing the test purpose formula φ^{TP} automaton transition to its *End* location. For example, in Fig. 15, the automaton that implements root formula *P*, synchronizes its transition to the location *End* with *StopWatch* transition to the location *Passed* via upwards channel *Done_P*.

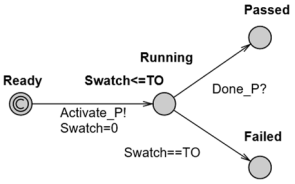


Figure 15 – Test Stopwatch automaton.

Another extension to *the supervisor model* is the capability of recording the test diagnostic information. For that each sub-formula of the test purpose specification formula φ^{TP} is indexed according to its position in the parsing tree of φ^{TP}. A diagnostic array *D* of type Boolean and of the size equal to the number of sub-formulas in φ^{TP} is defined in the model. The initial valuation of *D* sets all its elements to *false*. Whenever a model fragment that corresponds to a sub-formula reaches its end state (that is sub-formula satisfaction state), the element in *D* that corresponds to that sub-formula is set to *true*. It means that if the test passes, all elements of *D* are updated to *true*. Otherwise, in case

the test fails, those elements of D remain *false* which correspond to the sub-formula automata which conditions were not satisfied or reached by the test model run. The updates $D[i]$:= *true* of array D elements, where i is the index of the sup-formula automaton $M^{op}{}_i$, are shown on the edges that enter their *End* locations. The expression automata $M^{op}{}_i$ and their mapping to composition wrapping are shown in Fig. 14.

The test model construction steps can be summarized now as follows:

1. the test purpose is specified as a TDL^{TP} expression φ^{TP};
2. trapsets TS_1,…, TS_n are extracted from φ^{TP} and the ground level TCIs are labelled with elements of non-intersecting trapsets;
3. the parse tree of the TDL^{TP} expression φ^{TP} is analysed and each of its sub-formula operator op_i is mapped using the mappings $M1$ to $M10$ to the automaton template $M^{op}{}_i$ that corresponds to the sub-formula operation;
4. the labelling of M^{SUT} with traps is simplified by applying rules in Section 4.6, and M^{SUT} linked with sub-formula automata $M^{op}{}_i$ via wrapping construct that provides channels for signalling about reaching the state, where sub-formula are satisfied;
5. finally, the extension for collecting diagnostics is added to automata $M^{op}{}_i$ and the root formula automaton is composed with *Stopwatch* automaton M^{SW} which decides on the test pass or fail.

The total test model is synchronous parallel composition of component models $M^{SUT} || M^{SW} ||_i M^{op}{}_i$.

9. CASE STUDY

To demonstrate the usability of TDL^{TP} the TTU100 satellite testing case study has been chosen. The objective of the TTU100 project is to build a space system consisting of a 1U (10 cm x 10 cm x 10 cm) nanosatellite and a ground station, where mission planning and mission control software for scientific experiments is installed. The TTU100 system consists of a Ground Segment and a Space Segment. The Ground Segment communicates, stores and processes data aquired from satellite. The Space Segment is nanosatellite on Earth's Sun Synchronous Orbit (650km altitude). The satellite onboard system consists of smart electrical power supply (EPS), attitude determination and control system (ADCS), on-board computer (OBC), communication system (UHF band, Ku-band) and camera and optics payload.

For TDL^{TP} usability demonstration smart EPS subsystem is selected as a SUT. The test purpose is specified for a test case which demonstrates the TDL^{TP} capability to express combinations of multiple coverage criteria in a single test case. From TDL^{TP} expressions the test models are constructed and the test sequences generated using Uppaal model checker. The section is concluding with comparison of the tests generated with the methods presented in the paper and with those available using ordinary TCTL model checking.

9.1 SYSTEM UNDER TEST MODELLING

EPS receives commands from other system components to change its operation mode and respond with its status information. In the integration level test model we abstract from the concrete content of the commands and responses and describe its interface behavior in response to input commands.

EPS is sampling its input periodically with period 20 time units. EPS wakeup time when detecting a new input command can vary within interval [15, 20] time units after previous sampling. After wakeup it is ready to receive incoming commands. Due to internal maintenance procedures of EPS some of the commands when sent during self-maintenance can be ignored, and need to be repeated later. The command processing after its successful receive takes at most 20 time units. Thereafter, the validity of the command is checked using CRC error-detecting code. If the error is detected the error report will be sent back to EPS output port in *o_response* message. If the received command data is correct, the command is processed and its results returned in the outgoing *o_message*. Since EPS internal processing time is negligible compared to that of input sampling period and wakeup time, all the other locations except *start* and *commandCreated* are modelled as committed locations. The model M^{SUT} of the EPS is depicted in Fig. 16.

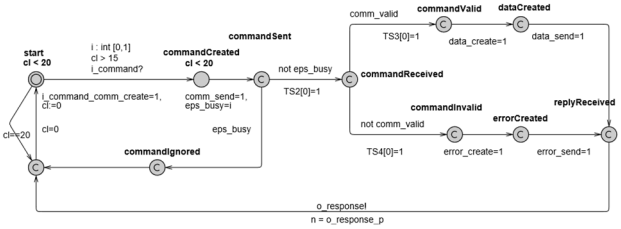


Figure 16 – The model M^{SUT} of the Electrical Power Supply subsystem.

9.2 TEST PURPOSE SPECIFICATION

The goal of test case is to show that after invalid command has been received the valid command can be received correctly and responded with acknowledgement. We specify the test purpose in TDL^{TP} as formula

$$A(TS2;TS4) \sim> E(TS2;TS3), \quad (25)$$

which expresses that all transition pairs labelled with traps of TS_2 and TS_4 must lead to some pair of transitions labelled with traps of trapsets TS_2 and TS_3.

9.3 LABELLING OF M^{SUT}

The labelling of M^{SUT} starts from the ground level trapsets TS_2, TS_3 and TS_4 of the formula (25). These traps guide branching conditions to be satisfied in the test scenario. The labelling is shown in Fig. 16.

Second level labelling results in applying trapset operation next ';' for pairs *TS2;TS3* and *TS2;TS4* which presumes introducing auxiliary variables *fl23* and *fl24* to identify occurrence of traps of TS_3 and TS_4 right after traps of TS_2. Since *TS2;TS3* and *TS2;TS4* are arguments of the upper '*forall*' and '*exists*' formula their occurrence should be signaled respectively to '*forall*' and '*exists*' automata. For this purpose additional committed locations and edges with upwards channels *ch_TS23* and *ch_TS24* are introduced in Fig. 17.

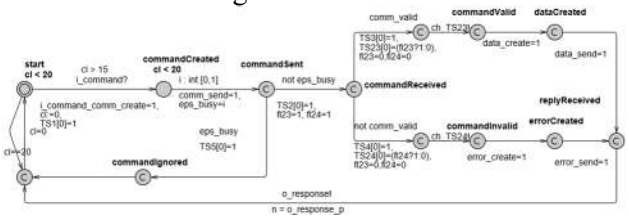


Figure 17 – Marking *TS2;TS3* and *TS2;TS4* trapsets

9.3 TEST MODEL CONSTRUCTION

When moving upwards in the parse tree of formula (25) the next operators that have *TS2;TS4* and *TS2;TS3* in arguments are forall *A(TS2;TS4)* and exists *E(TS2;TS3)* which automata are depicted in Fig. 18.

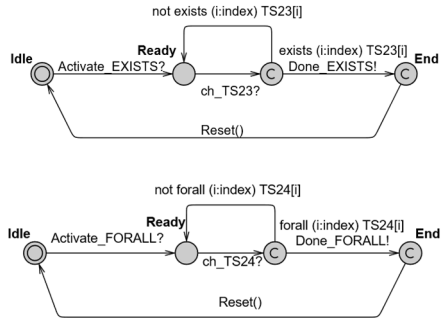


Figure 18 – a) automation that recognizes A(TS2;TS4); b) automation that recognizes E(TS2;TS3) respectively

The root operator in the formula (25) is '*leads to*' the arguments of which are *A(TS2;TS4)* and *E(TS2;TS3)*. The automaton that recognizes *A(TS2;TS4)* ~> *E(TS2;TS3)* is depicted in Fig. 19.

The full test model for generating test sequences of test scenario *A(TS2;TS4)* ~> *E(TS2;TS3)* is composed of automations shown in Fig. 17, Fig. 18, Fig.19 and Fig. 20.

Figure 19 – Recognizing automaton of A(TS2;TS4) ~> E(TS2;TS3)

Figure 20 – Automaton for Environment and StopWatch of Test model for implementing test scenario A(TS2;TS4) ~> E(TS2;TS3)

9.4 GENERATING TEST SEQUENCES

The test sequences of the SUT model M^{SUT} shown in Fig. 16 and of the scenario *A(TS2;TS4)* ~> *E(TS2;TS3)* are generated by running the model checking query *E<> StopWatch.Pass*. There are three options of selecting the trace for test - *shortest*, *fastest*, or *some*. The trace generated with model checking option *shortest* is shown in the Fig.21.

Simulation Trace

(-, start, Idle, Idle, Idle, Ready)
Activate_LEADSTO: StopWatch → P_LEADSTO_Q
(-, start, Idle, Idle, -, Running)
Activate_EXISTS: P_LEADSTO_Q → FORALL_TS24
(-, start, Ready, Idle, Ready_for_P, Running)
i_command: Environment[1] → MCS_EPS
(-, commandCreated, Ready, Idle, Ready_for_P, Running)
MCS_EPS[0]
(-, commandSent, Ready, Idle, Ready_for_P, Running)
MCS_EPS
(-, commandReceived, Ready, Idle, Ready_for_P, Running)
MCS_EPS
(-, -, Ready, Idle, Ready_for_P, Running)
ch_TS23: MCS_EPS → FORALL_TS24
(-, commandValid, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, dataCreated, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, replyReceived, -, Idle, Ready_for_P, Running)
o_response: MCS_EPS → Environment
(-, -, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, start, -, Idle, Ready_for_P, Running)
Done_EXISTS: FORALL_TS24 → P_LEADSTO_Q
(-, start, End, Idle, -, Running)
FORALL_TS24
(-, start, Idle, Idle, -, Running)
Activate_FORALL: P_LEADSTO_Q → EXISTS_TS23
(-, start, Idle, Ready, Ready_for_Q, Running)
i_command: Environment[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready_for_Q, Running)

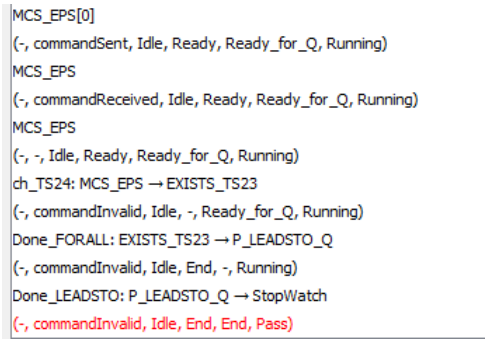


Figure 21 – Test Case test sequence

The lenght of the trace generated by using TDL^{TP} is 22 transitions and the average lenght generated using ordinary TCTL model checking is 50 transitions.

9. CONCLUSION

In this paper high level test purpose specification language TDL^{TP}, its syntax and semantics have been defined for model-based testing of time critical systems. Based on the semantics proposed in this work a mapping from TDL^{TP} to Uppaal TA formalism has been defined. The mapping is used for automatic construction of test models that are composition of a SUT model and the tester model derived from the test purpose specification in TDL^{TP}. Practical side effect of this is the diagnosis capability enabling tracing back the specification sub-formulae which violation by SUT behavior causes test fail. The application of TDL^{TP} based test generation approach on the TTU100 satellite power supply system case study confirmed our expectations that complex multi-purpose test goals can be specified in compact and comprehensible way saving from time consuming and error prone manual test scripting. Future study is needed to evaluate the capability of TDL^{TP} to specify and provide efficient online interpretation algorithms of non-linear dynamics phenomena such as bifurcation and chaotic behavior of complex systems.

ACKNOWLEDGEMENT

This research was partially supported by the Estonian Ministry of Education and Research institutional research grant no IUT33-13.

10. REFERENCES

[1] *Object Management Group (OMG)*: CCDL whitepaper, Razorcat Technical Report, January 2014. [Online]. Available: http://www.razorcat.eu/PDF/Razorcat_Technical_Report_CCDL_Whitepaper_02.pdf.

[2] *Robot Framework*. [Online]. Available at: https://robotframework.org.

[3] T. Pajunen, T. Takala, and M. Katara, "Model-based testing with a general purpose keyword-driven test automation framework," *Proceedings of the 4th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST 2012*, 2011, pp. 242–251.

[4] A. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, Y. Fusero, and M. Schieber, "A meta-model for tests of avionics embedded systems," Proceedings of the 1st Int. Conf. on Model-Driven Engineering and Software Development MODELSWARD 2013, SciTePress, 2013, pp. 5–13.

[5] J. Grossmann and W. Müller, "A formal behavioral semantics for testml," *Proceedings of the 2nd Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA'2006*, 2006, pp. 441–448.

[6] *ISO: Road Vehicles – Open Test Sequence Exchange Format, Part 3: Standard Extensions and Requirements*, International ISO Multipart Standard No. 13209-3, 2017.

[7] *ISO/IEC: Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework, Part 1: General Concepts*, International ISO/IEC Multipart Standard No. 9646, 1994/S1998.

[8] *ITU Recommendation Z.120: Message Sequence Chart (MSC)*, 02/11. [Online] Available at: http:// www.itu.int/rec/T-REC-Z.120-201102-I/en.

[9] *ITU Recommendation Z.120: Annex B: Formal Semantics of Message Sequence Chart (MSC)*, 04/98. [Online] Available at: http://www.itu.int/rec/T-REC-Z.120-199804I!AnnB/en.

[10] ETSI: TDL. [Online] Available at: http:// www.etsi.org/deliver/etsi_tr/103100_103199/103119/01.01.01_60/tr_103119v010101p.pdf.

[11] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," *Proceedings of the 3rd ACM WS on Advances in Model Based Testing, A-MOST 2007, co-located with the ISSTA 2007*, 2007, pp. 95–104.

[12] P. Makedonski et al., "Test descriptions with ETSI TDL," *Software Quality Journal*, vol. 27, issue 2, pp. 885-917, June 2019. DOI: https://doi.org/10.1007/s11219-018-9423-9.

[13] *ETSI ES 202 553: Methods for testing and specification (mts)*, TPLan: A notation for expressing Test Purposes, v1.2.1. ETSI, Sophia-Antipolis, France, June 2009.

[14] A. David, K. G. Larsen, S. Li, and B. Nielsen, "A game-theoretic approach to real-time system testing," *Proceedings of the ACM International Conference on Design, Automation and Test in Europe DATE'2008*, 2008, pp. 486–491.

[15] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen, "Uppaal SMC tutorial," *STTT*, vol 17, no 4, pp. 397–415, 2015.

[16] J. Bengtsson, W. Yi, "Timed automata: Semantics, algorithms and tools," in: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Lectures on Concurrency and Petri Nets: Advances in Petri Nets, Lecture Notes in Computer Science*, Springer, Heidelberg, 2004, vol. 3098, pp. 87–124.

[17] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, "Testing real-time systems using UPPAAL," in: R. Hierons, J. Bowen, M. Harman (Eds.) *Lecture Notes in Computer Science*, Springer, Heidelberg (2008), vol. 4949, pp. 77-117.

[18] G. Behrmann, A. David, K. G. Larsen, "A tutorial on UPPAAL," in: Bernardo, M., Corradini, F. (eds.) *Formal Methods for the Design of Real-Time Systems. Lecture Notes in Computer Science*, Springer, Heidelberg, 2004, vol. 3185, pp. 200-236.

[19] J. Vain, M. Kääramees, M. Markvardt, "Online testing of nondeterministic systems with reactive planning tester," in: Petre, L., Sere, K., Troubitsyna, E. (eds.) *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*, IGI Global, Hershey, 2012, pp. 113-150.

[20] C. Arilo, D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering ASE'2007 (WEASELTech'07)*, 2007, pp. 31-36. http://dx.doi.org/10.1145/1353673.1353681

[21] C. Baier and J.-P. Katoen, *Principles of Model Checking*, MIT Press, 2008.

[22] Yu. Kolokolov and A. Monovskaya, "A practice-oriented bifurcation analysis for pulse energy converters, Part 4: Emergency forecasting," int. journal of bifurcation and chaos, vol. 28, no. 12, article 1850152, 2018.

***MSc. Evelin Halling**, has received MSc degree in Computer Science from Tallinn University of Technology in 2011. Currently, PhD student at the Dep. of Software Science, Tallinn University of Technology. Her research interests include formal methods, software testing, model-based testing, robotics and machine learning.*

***Dr. Jüri Vain,** graduated in System Engineering from Tallinn Polytechnic Institute, Estonia in 1979. He received his PhD in Computer Science from the Estonian Academy of Sciences in 1987. Currently, he is Prof. of Computer Science at the Dep. of Software Science, Tallinn University of Technology. His research interests include formal methods, model-based testing, cyber physical systems, human computer interaction, autonomous robotics, and artificial intelligence.*

***Dr. Artem Boyarchuk** is an Associate Professor at the Computer Systems, Networks and Cybersecurity Department of the National aerospace university n. a. N. E. Zhukovsky "Kharkiv Aerospace Institute". He has received M.S. degree in Computer Engineering from the "Kharkiv Aviation Institute" in 2005 and defended a PhD in 2012. His expertise is in models and methods for availability assessment of service-oriented infrastructures for business-critical applications.*

***Dr. Oleg Illiashenko** is a senior lecturer at the Computer Systems, Networks and Cybersecurity Department of the National aerospace university n. a. N. E. Zhukovsky "Kharkiv Aerospace Institute". He has received M.S. degree in Computer Engineering from the Kharkiv Aviation Institute in 2012 and Sp.Ed. in Information and communication systems security from the Kharkiv National University of Radio Electronics, Ukraine in 2014 and defended a PhD in 2018. His expertise is in models, methods and instrumentation tools for information security and cyber security assessment, evaluation and assurance of cyber security of software and hardware, dependability and resilience of embedded, web, cloud and IoT systems.*