Sequential Composition of BDD Transition Systems for Model-Based Testing *

Tannaz Zameni[1][0009−0000−6733−5612], Petra van den Bos[1][0000−0002−9212−1525], Johan Foederer[2][0009−0009−7999−5365], and Arend Rensink[1][0000−0002−1714−6319]

[1] University of Twente, Enschede, The Netherlands
{t.zameni,p.vandenbos,arend.rensink}@utwente.nl
[2] TOPIC Embedded Systems, Best, The Netherlands
johan.foederer@topic.nl

Abstract. This paper presents a compositional approach to model-based test derivation in Behavior-Driven Development (BDD). In BDD, system behavior is specified through *scenarios* written in natural language. For each scenario, a test case can be derived. However, such test cases do not cover the integration of multiple behaviors, while that is where potential faults may very well occur. To counter this, we introduce a formal composition operator for *sequential composition*, which integrates the individual BDDs while preserving their test coverage. We also report on a prototype tool that integrates model-based features into an existing testing framework that supports BDD-based test derivation. We show the feasibility and advantages of our approach by applying the prototype to a real-world case study.

Keywords: BDD Transition Systems · Symbolic Transition Systems · Composition · Behavior-Driven Development · Model-Based Testing.

1 Introduction

Testing is the most widely applied validation method in software development. However, the effort, time, and cost spent on testing cannot keep up with the large and complex systems being developed nowadays. Model-based testing (MBT) has been proposed as an approach to automate part of this process; see, e.g., [16]. In MBT, tests are not only executed automatically but also generated automatically from a *model* that specifies the expected behaviors of the system. Automatic test generation algorithms generate diverse test cases to find many more (potential) bugs than could be found by writing test cases manually. However, one of the factors hindering the widespread adoption of MBT is that a model has to be provided before MBT can be applied. The need to create and maintain such models, typically in unfamiliar languages and formalisms for software developers, in many circumstances negates the potential benefits.

* Artifact available at https://doi.org/10.5281/zenodo.14892704.

Behavior-driven development (BDD) is a popular agile software development approach. As part of this approach, behaviors are specified in natural language scenarios. These scenarios are formulated as a result of the understanding reached between the involved stakeholders; typically a product owner, developer, and tester of an agile team. They are often written in a structured format, using the Given-When-Then style of the Gherkin notation [12,4]. Such a BDD scenario expresses a behavior in three steps: *Given* a precondition for the required system state, *When* an action is performed on or by the system, *Then* specified actions and resulting state are expected. There exist BDD tools, like Cucumber [5] and SpecFlow [15], that allow users to manually implement how each step in a scenario is performed in the system, after which they automatically compose the implemented steps into test cases. Though the required manual effort and the unspecified semantics of these tools mean that they do not meet the standards of MBT, their adoption in practice shows that BDD scenarios are an interesting candidate to fill the modeling gap.

In [18], we outlined an approach to translate BDD scenarios to models suitable for formal test generation, called BDD Transition Systems (BDDTSs). Being derived from a single scenario, each BDDTS comprises a single sequence of actions and checks. Although the generated test cases adhere to the formal semantics of BDDTS, they remain singular and do not integrate behaviors, just like the tests generated from BDD tools. To really harvest the power of MBT, in [19] we sketched the idea to compose individual BDDTSs into more complex models. In this paper, we formally define *sequential composition* on (an improved version of) BDDTSs, i.e., invoking them one after the other. This is valid if the *Then*-step of one scenario (called the "predecessor" below) implies the *Given*-step of another scenario.

The main strength of this composition lies in its *integration* capability: from the composed model, we can derive test cases that cover the integration of scenarios, i.e., whether they work together as expected. This has three advantages. Firstly, in standard BDD-based testing, achieving the desired state for a *Given* step may require complex *setup* and *teardown* phases, adding overhead and imposing a fixed test execution order. Sequential composition reduces spurious setups and teardowns, and enables the generation of varied, integrated test sequences. Secondly, testers might make assumptions in implementing tests, that are only true when tests are run in a certain order. The varied combinations of tests derived from a composed model help reveal incomplete test implementations, thereby improving test quality. Last but not least, different features of the system (embodied in different BDD scenarios) may interact in unexpected ways that are difficult to predict and detect. Scenario composition helps to identify issues missed by the standard BDD approach.

Results of the paper. This paper formally defines and practically applies sequential composition, as well as the corresponding test cases, and demonstrates their application through a practical case study. The key contributions are:

1. **Formal definition of sequential composition.** We formalize the idea suggested in [19], where the sequential composition was proposed but lacked a formal definition.
2. **Proof of soundness and completeness.** We show that sequential composition preserves the *potential* test coverage; that is, the test suite derived from the individual scenarios *potentially* tests for the same faults as that derived from their composition – namely, if the tests would be applied exhaustively, meaning that each test is repeated for all possible setups that bring the system to a state satisfying its *Given* clause.
3. **Formal definition of test case derivation.** In [18] we discussed three potential methods to convert BDDTSs to test cases. Here, we adopt and formalize one, based on retrieving variable values in the System under Test (SuT) and in output guards, as well as satisfying the initial input guard (while others are validated through composition).
4. **Practical application and results.** We show the practical feasibility of our approach by developing an extension of Robot Framework [13] with MBT and applying it to a real-world case study: a distributed system that controls the lamps on airfields. The tool and case study are available in the artifact[17].

The paper outline is as follows. We introduce our case study in section 2. We then recall BDD transition systems (section 3), and formally define how to convert them to test cases (section 4). After, we formally define the sequential composition operator and its soundness and completeness properties in section 5. Then we show the results of the case study using our tool in section 6. Finally, we discuss related work (section 7), conclusions, and future work (section 8).

2 Introduction to the Case Study

The case study concerns the software system of the company TKH Airport Solutions. The software controls airfield lamps, a so-called Airfield Ground Lighting system (AGL). Lamps are grouped into AGL functions, which each correspond to something meaningful to the Air Traffic Controller, like a certain runway or taxiway. Lamps are locally powered and managed by basestations. Typically, multiple basestations are used to operate a single AGL function.

The main purpose of the feature specified by our scenarios is to demonstrate that if maintenance is required for a specific basestation, it can be carried out in isolation: only the AGL function it serves is affected, and other parts of the airfield stay operational. There are three roles involved in the BDD scenarios:

- *Air Traffic Controller* (ATC), responsible for operating AGL functions. They do not need to be aware of basestations.
- *Maintenance Manager* (MM), responsible for basestation maintenance and their impact on AGL functions. They can release a basestation from operation when an AGL function is cleared by the ATC and no longer in use.

– *Maintenance Engineer* (ME), responsible for installing and maintaining basestations and their connected lamps. They access a basestation airside, where the airplanes operate, once it has been released by the MM.

ATC and MM use the same interface to interact with the system, while ME uses a different interface. So unlike the BDD scenarios, we make no difference between ATC and MM in the derived models and test implementations.

Below we provide a selection of the BDD scenarios we used in the case study. The Background scenario ensures we have the correct system configuration, the others describe three relevant behaviours. For readability, we slightly simplified the phrasing of the original scenarios for this paper.

Scenario 0 Background
- ***Given*** Given airfield of airportA is operated by CEDD-AGL system
  And basestation07 serves the runway
  And basestation13 serves the taxiway
  And basestation14 serves the taxiway

Scenario 1 The runway stays open during maintenance on the taxiway
- ***Given*** Given the runway is under the operation of the Air Traffic Controller
  And the taxiway is under the operation of the Air Traffic Controller
- ***When*** the Maintenance Manager confirms that only the taxiway is affected by basestation13
  And the Maintenance Manager releases basestation13 from operation
  And the Maintenance Engineer puts basestation13 into maintenance
- ***Then*** the Maintenance Engineer has basestation13 in maintenance
  And the runway is (still) under the operation of the Air Traffic Controller

Scenario 2 A single basestation is handed over for maintenance
- ***Given*** Basestation13 is under the control of the Air Traffic Controller
- ***When*** the Maintenance Manager releases basestation13 from operation
  And the Maintenance Enigneer puts basestation13 into maintenance
- ***Then*** the Maintenance Engineer has basestation13 in maintenance

Scenario 3 a basestation is handed back without configuration change
- ***Given*** the Maintenance Engineer has basestation13 in maintenance
- ***When*** the Maintenance Engineer releases basestation13 from maintenance
  And the Maintenance Manager puts basestation13 into operation
- ***Then*** the mode of basestation13 is operational
  And the taxiway is under the operation of the Air Traffic Controller

3 BDD Transition Systems

Before introducing BDD Transition Systems, we need some auxiliary notation. We write $f : X \to Y$ or $f \in Y^X$ for a total function f from domain X to codomain Y, and $f : X \hookrightarrow Y$ if f is partial. In the latter case, $f \downarrow x$ denotes that f is defined for x, and $f \uparrow x$ that it is undefined. $f\restriction_Z$ denotes the restriction of f to the subdomain $Z \subseteq X$, $f \uplus g$ the union of (compatible) functions.

In this paper, we often use *transition relations*, which are always defined as $\to \subseteq Q \times A \times Q$ for a set Q of states (sometimes called locations) and a set A of labels. For this, we adopt some common notations. In particular, $q_0 \xrightarrow{a_0 \cdots a_{n-1}} q_n$ denotes $(q_i, a_i, q_{i+1}) \in \to$ for $0 \leq i < n$, and $q \xrightarrow{\sigma}$ for $\sigma \in A^*$ means that there is some q' such that $q \xrightarrow{\sigma} q'$. A state q is called a *sink*, denoted $\mathit{Sink}(q)$, if $\nexists a : q \xrightarrow{a}$, and the *traces* of q are given by $\mathit{traces}(q) = \{\sigma \in A^* \mid q \xrightarrow{\sigma}\}$.

In several cases, the set Q will be partitioned into $Q = Q^\circ \cup Q^\bullet$ of *open* and *closed* elements to further distinguish between actions that lead to failing

tests (from closed locations) and the ones that lead to inconclusive or pass (from open locations). We call such sets $\overset{\circ}{\bullet}$-*natured*. For a $\overset{\circ}{\bullet}$-natured set X we will use $\mathsf{N} : X \to \{\circ, \bullet\}$ to retrieve the nature (hence $x \in X^{\mathsf{N}(x)}$ for all x).

The core models of this paper are *BDD Transition Systems* (BDDTSs). These are *symbolic* in the sense that the labels, as well as some other components, are built using syntactic *terms*, e.g. $\mathsf{x} + 1$ or $\mathsf{a} \wedge \mathsf{b}$, the value of which will be computed later. We use $\mathcal{T}_s(V)$ to denote the set of s-typed terms over some set of variables V; omitting s gives us *all* terms. We also use $\mathcal{I}(G, IV)$ to denote *interactions*, which are pairs $(g, i\bar{v})$ of *gates* $g \in G$ and sequences of *interaction variables* $i\bar{v} \in IV^*$. Every gate g has a fixed set of interaction variables. Terms and interactions are typed in the standard way; we omit the details here.

Definition 1. *A* BDD Transition System *(BDDTS) is a tuple* $\mathcal{B}$ *consisting of* $\langle L, V, G, \to, il, IG, OG \rangle$*, where*

- L is a $\overset{\circ}{\bullet}$-natured set of locations*;*
- V is a set of variables, partitioned into MV *(model variables),* CV *(context variables)* and IV *(interaction variables).*
- G is a set of gates, partitioned into G_i *(input gates)* and G_o *(output gates);*
- $\to \subseteq L \times \mathcal{I}(G, IV) \times \mathcal{T}_{\mathsf{Bool}}(V) \times \mathcal{T}(V)^{MV} \times L$ is a switch relation*;*
- $il \in L^\circ$ is the initial (open) location*;*
- $IG \in \mathcal{T}_{\mathsf{Bool}}(MV \cup CV)$ is the *input guard;*
- $OG : L \hookrightarrow \mathcal{T}_{\mathsf{Bool}}(MV \cup CV)$ is a partial map from locations to *output guards.*

We use $L_{\mathcal{B}}$, $V_{\mathcal{B}}$ etc. to denote the components of a BDDTS $\mathcal{B}$; for $\mathcal{B}_i$, we further abbreviate $L_{\mathcal{B}_i}$, $V_{\mathcal{B}_i}$ etc. to L_i, V_i. The components of a switch $t \in \to$ are denoted sl_t (the *source location*), $\alpha_t = (g_t, i\bar{v}_t)$ (the *interaction*), ϕ_t (the *switch guard*, a boolean term), a_t (the *assignment*, a function from MV to terms) and tl_t (the *target location*); for a switch denoted t_i, we write sl_i, α_i etc.

A BDDTS is *instantiated* by assigning values to all *location variables* $LV = CV \cup MV$. After instantiation, the context variables (CV) never change, in contrast to the model variables (MV). The intuition behind a switch t is that it encodes the possibility to move from the source location sl_t to the target location tl_t while interacting with the environment via α_t, but only if the switch guard ϕ_t holds. Upon reaching tl_t, all model variables receive a new value, as dictated by a_t. Upon reaching a *goal location* l, being a location on which OG is defined, the output guard $OG(l)$ is supposed to hold.

$\mathcal{B}$ is called *well-formed* if it satisfies the following additional conditions:

- *Switching is deterministic*: for distinct $t_1, t_2 \in \to$, if $sl_1 = sl_2$ and $\alpha_1 = \alpha_2$ then $\phi_1 \wedge \phi_2 \equiv \mathbf{false}$.
- *Closed locations only have output switches*: $t \in \to$ with $sl_t \in L^\bullet$ implies $g_t \in G_o$.
- *Sink locations are always open*: *Sink*(q) implies $q \in L^\circ$.

We require all BDDTSs to be well-formed, and henceforth tacitly assume this.

The $\overset{\circ}{\bullet}$-nature of a location determines how to treat, during testing, an output action of the SUT that deviates from the outgoing switches specified for that

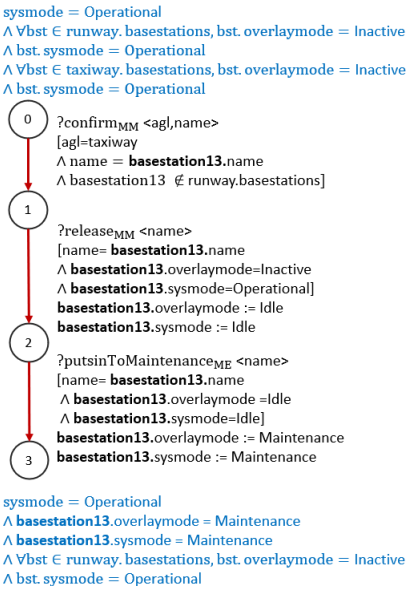


Fig. 1: BDDTS $\mathcal{B}_1$ corresponding to scenario 1: AGL function handover

location. For an open location, such a deviation is taken to mean that the SuT was inadvertently not in the expected state; the test is then marked as *inconclusive*. In a closed location, on the other hand, a deviation is regarded as a fault of the system; the test is marked as *failed*.

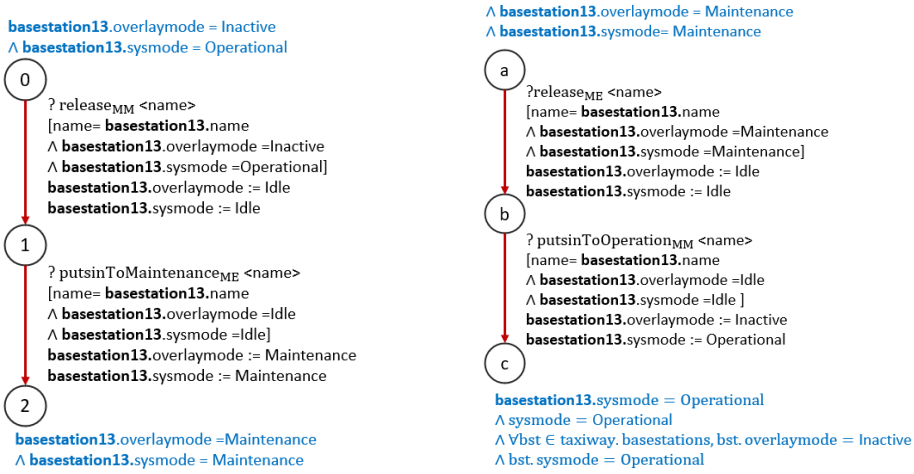
Example 1. Figure 1 shows the BDDTS for **Scenario 1**. We extract model elements from the textual scenario through stakeholder discussions and the method in [18]. The BDDTS elements are explained below:

- $L = L^{\circ} = \{0, 1, 2, 3\}$ and $il = 0$.
- $V = MV \cup CV \cup IV$ with $MV = \{\mathsf{basestation13}\}$, $CV = \{\mathsf{sysmode}, \mathsf{runway}, \mathsf{taxiway}\}$ and $IV = \{\mathsf{agl}, \mathsf{name}\}$
- $G = G_i = \{?\mathsf{confirm}_{\mathsf{MM}}, ?\mathsf{release}_{\mathsf{MM}}, ?\mathsf{putsinToMaintenance}_{\mathsf{ME}}\}$
- The switches are derived from the *When* step. For instance, the first switch $t = (0, ?\mathsf{confirm}_{\mathsf{MM}}\ \mathsf{agl}\ \mathsf{name}, \mathsf{agl} = \mathsf{taxiway}, \mathsf{name} = \mathsf{basestation13.name}, id, 1)$
- IG, derived from the *Given* step, is shown in blue on location 0.
- OG, derived from the *Then* step, is only defined for location 3 (in blue).

The BDDTSs for **Scenario 2** and **Scenario 3** are shown in Figure 2a and 2b.

4 From BDDTSs to Test Cases

We now come to the first contribution: the formal conversion of BDDTSs to test cases. This follows the steps in Figure 3, and corresponds to one of the (informally discussed) proposals in [18].



(a) $\mathcal{B}_2$: single handover (b) $\mathcal{B}_3$: single handback

Fig. 2: BDDTSs corresponding to scenarios of section 2

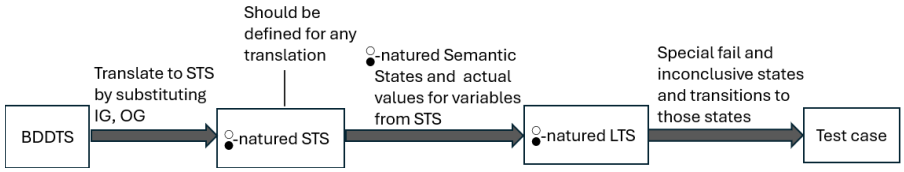


Fig. 3: BDDTS to Test Cases Translation Steps

We first define a translation from BDDTSs to *Symbolic Transition Systems* in which the input and output guards are implemented: the former by instantiating location variables appropriately, whereas the latter are replaced by sequences of *check* and *retrieve* switches.

Definition 2. *A* Symbolic Transition System *(STS) is given by a tuple* $\mathcal{S} = \langle L, V, G, \rightarrow, il, ini \rangle$*, where the first five components are identical to those of a BDDTS with the proviso that* $CV = \emptyset$*, and* $ini \in \mathcal{T}(\emptyset)^{LV}$ is the initial assignment *of the location variables.*

The input guard of a BDDTS can be represented in an STS with an $ini \in \mathcal{T}(\emptyset)^{LV}$ that satisfies the input guard, denoted as $[\![IG]\!]_{ini} \equiv \mathbf{true}$. For the output guards, the translation to STS is more involved. We assume that for all $v \in LV$, there exist special input gates $\mathsf{check}_v \in \mathcal{G}$ and output gates $\mathsf{retrieve}_v \in \mathcal{G}$ with interaction variable iv_v. (For convenience, since there is no risk of confusion, we actually denote iv instead of iv_v.) A $?\mathsf{check}_v$-labelled switch asks the SuT to produce the current value of v; the SuT is expected to respond immediately by $!\mathsf{retrieve}_v$ iv

with iv instantiated to that value. If v is a model variable, the switch guard immediately compares iv to v; if v is a context variable, the value of iv is assigned to v instead.[3] To define this formally, let $v_1 \cdots v_n$ be the vector of variables occurring in $OG(gl)$, in their order of occurrence. As is shown in Figure 4, we introduce new locations $\hat{L}_{gl} = \{chl_{gl,i} \mid 1 \le i \le n\} \cup \{rtl_{gl,i} \mid 1 \le i \le n\}$ (disjoint from L) for check and retrieve locations, with $\circ\!\!\bullet$-nature defined by

$$\mathsf{N} : l \mapsto \begin{cases} \mathsf{N}(gl) & \text{if } l = rtl_{gl,n} \\ \circ & \text{if } l = rtl_{gl,i} \text{ for } 1 \le i < n \\ \bullet & \text{if } l = chl_{gl,i} \text{ for } 1 \le i \le n \end{cases}$$

In the following switches between the new locations $\hat{L}_{gl}$, let $rtl_{gl,0} = gl$. Besides check and retrieve switches CH_{gl} and RT_{gl}, we need *connecting* switches CN_{gl} from $rtl_{gl,n}$ for every outgoing switch of gl (if any). Note that the output guard of goal location gl is turned into a switch guard of the last retrieve switch. (We write $t[x/y]$ with $t \in \mathcal{T}(X)$ and $x, y \in X$ for the replacement of all y by x in t.)

$$CH_{gl} = \{(rtl_{gl,i-1}, ?\mathsf{check}_{v_i}, \mathbf{true}, id, chl_{gl,i}) \mid 1 \le i \le n\}$$
$$RT_{gl} = \{(chl_{gl,i}, !\mathsf{retrieve}_{v_i} iv, \phi_i, a_i, rtl_{gl,i}) \mid 1 \le i \le n\}$$
$$\text{where } \phi_i = \begin{cases} iv{=}v_i \wedge OG & \text{if } i = n \wedge v_i \in MV \\ OG[iv/v_i] & \text{if } i = n \wedge v_i \in CV \\ iv{=}v_i & \text{if } i < n \wedge v_i \in MV \\ \mathbf{true} & \text{otherwise} \end{cases} \quad a_i = \begin{cases} v_i := iv & \text{if } v_i \in CV \\ id & \text{otherwise} \end{cases}$$
$$CN_{gl} = \{(rtl_{gl,n}, \alpha, \phi, a, tl) \mid gl \xrightarrow{\alpha,\phi,a} tl\}$$

With these ingredients, we can define the derived STS of a given BDDTS.

Definition 3. *Let $\mathcal{B} = \langle L, V, G, \to, il, IG, OG\rangle$ be a BDDTS, and $ini \in \mathcal{T}(\emptyset)^{LV}$ an assignment such that $[\![IG]\!]_{ini} \equiv \mathbf{true}$. Then the* check-retrieve symbolic system *for $\mathcal{B}$ and ini is defined as $CRSS(\mathcal{B}, ini) = \langle L_c, V_c, G, \to_c, il, ini\rangle$ where $L_c = \{l \in L \mid OG \uparrow l\} \cup \bigcup\{\hat{L}_{gl} \mid OG \downarrow gl\}$, $LV_c = MV \cup CV$, and*

$$\to_c = \{t \in \to \mid OG \uparrow sl_t\} \cup \bigcup\{CH_{gl} \cup RT_{gl} \cup CN_{gl} \mid OG \downarrow gl\} \ .$$

For instance, Figure 4 shows the CRSS obtained from the BDDTS in Figure 2a (modelling **Scenario 2**). The bold green transitions in green starting from location 2 are check and retrieve switches for variable basestation13; there is no outgoing transition from $rtl_{2,1}$ because the goal location in Figure 2a was a sink.

The next step is to translate a CRSS into an $\circ\!\!\bullet$-natured LTS, sometimes called the *interpretation* of the STS [3]. From LTSs we are able to derive test cases.

Definition 4. *A* labelled transition system *is a tuple $\mathcal{L} = \langle Q, A, \to, q_0\rangle$ in which Q is a set of states with initial state q_0, A is a set of labels partitioned into input labels A_i and output labels A_o, and $\to \subseteq Q \times A \times Q$ is a transition relation.*

[3] Note that we do not assign values to context variables: in STSs, all location variables are model variables.

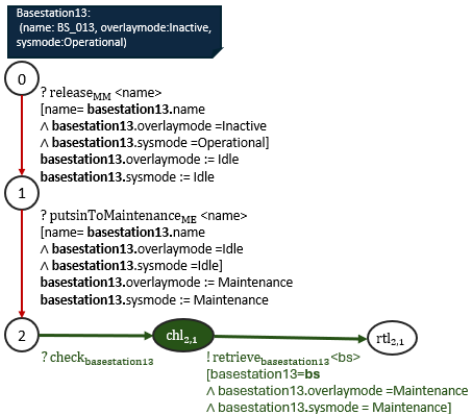


Fig. 4: CRSS of $\mathcal{B}_2$

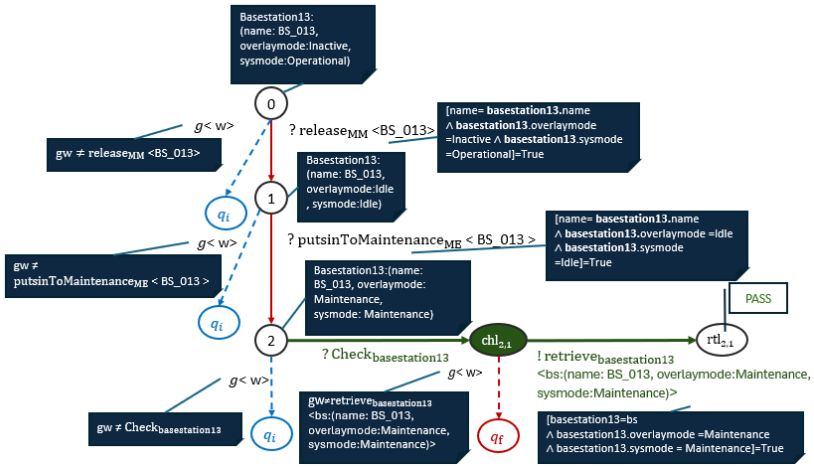


Fig. 5: Test Case for BDDTS $\mathcal{B}_2$

The interpretation of an STS involves a mapping to a semantic domain $\mathcal{U}$ of *values*, defined as usual; in particular, given a valuation $\vartheta : X \to \mathcal{U}$, every term $t \in \mathcal{T}(X)$ gives rise to a value $[\![t]\!]_\vartheta \in \mathcal{U}$, and every assignment $a \in T(X)^Y$ to a valuation $[\![a]\!]_\vartheta \in \mathcal{U}^Y$ such that $[\![a]\!]_\vartheta : v \mapsto [\![a(v)]\!]_\vartheta$ for all $v \in Y$. For ground terms, we may omit the valuation ϑ.

The states of an interpretation are called *semantic states*; the labels are *gate values*. Consider the STS $S = \langle L, V, G, \to, il, ini \rangle$.

– *Semantic states* are elements of $L \times \mathcal{U}^{LV}$. The nature of semantic states is taken from their locations: $\mathsf{N} : (l, \vartheta) \mapsto \mathsf{N}(l)$.

– *Gate values* are sequences $u = g\,\bar{w}$ where $g \in G$ and $\bar{w} \in \mathcal{U}^*$. Every such gate value u gives rise to a valuation $\vartheta_u \in \mathcal{U}^{IV}$.
– LTS transitions exist for every STS switch and every gate value that satisfies the switch guard. For a switch $t \in \rightarrow$ and gate value $u = g\,\bar{w} \in \mathcal{GU}$, we say that t is enabled for u in q, denoted $enab(q, u, t)$, if $[\![\phi_i]\!]_{\vartheta_u \uplus \vartheta_q} = \mathbf{true}$.
– The successor state of a u-labelled transition from q based on switch t is derived from t's target location and assignment: $succ(q, u, t) = (tl_t, [\![a_t]\!]_{\vartheta_u \uplus \vartheta_q})$.

Definition 5. *Let $\mathcal{S} = \langle L, V, G, \rightarrow, il, ini\rangle$ be an STS. The* interpretation *of $\mathcal{S}$ is defined as the LTS $[\![\mathcal{S}]\!] = \langle L \times \mathcal{U}^{LV}, G \times \mathcal{U}^*, \rightarrow', (il, [\![ini]\!])\rangle$ with*

$$\rightarrow' = \{(q, u, succ(q, u, t)) \mid q \in L \times \mathcal{U}^{LV}, u \in \mathcal{GU}, t \in \rightarrow, enab(q, u, t)\} \ .$$

Finally, we turn $\overset{\circ}{\bullet}$-natured LTSs into *test cases*, which are themselves LTSs in which particular (sink) states are identified as *fail* and *inconclusive*. This is the step in which the $\overset{\circ}{\bullet}$-nature comes into play.

Definition 6. *A* test case *is tuple $\langle Q, A, \rightarrow, q_0, q_{\mathbf{f}}, q_{\mathbf{i}}\rangle$ where $\langle Q, A, \rightarrow, q_0\rangle$ is an LTS and $q_{\mathbf{f}}, q_{\mathbf{i}} \in Q$ are special* fail *and* inconclusive *sink states, distinct from q_0.*

A test case TC is used as follows: an SuT (in a given state) is said to *fail* TC when its observable behaviour σ is such that $q_0 \xrightarrow{\sigma} q_{\mathbf{f}}$; and TC is *inconclusive* if $q_0 \xrightarrow{\sigma} q_{\mathbf{i}}$. If, finally, $q_0 \xrightarrow{\sigma} q$ for some sink state $q \notin \{q_{\mathbf{f}}, q_{\mathbf{i}}\}$, the SuT *passes* TC.

To obtain a test case from a $\overset{\circ}{\bullet}$-natured LTS $\langle Q, A, \rightarrow, q_0\rangle$, we add fresh states $q_{\mathbf{f}}, q_{\mathbf{i}}$ and, moreover, new transitions for every existing non-sink state $q \in Q$ and every output label for which q does not have an outgoing transition. The target state for such a new transition is $q_{\mathbf{f}}$ if $\mathsf{N}(q) = \bullet$ and $q_{\mathbf{i}}$ if $\mathsf{N}(q) = \circ$, reflecting the intuition that an unexpected output action from a closed state is an error, whereas from an open state, it is outside the scope of this test. A sink state that is neither $q_{\mathbf{f}}$ nor $q_{\mathbf{i}}$ indicates a pass. Formally:

Definition 7. *Let $T = \langle Q, A, \rightarrow, q_0\rangle$ be an $\overset{\circ}{\bullet}$-natured LTS and $q_{\mathbf{f}}, q_{\mathbf{i}} \notin Q$. The test case for T is defined by $TC(T) = \langle Q \cup \{q_{\mathbf{f}}, q_{\mathbf{i}}\}, A, \rightarrow', q_0, q_{\mathbf{f}}, q_{\mathbf{i}}\rangle$ such that*

$$\begin{aligned} \rightarrow' = \rightarrow \cup\ & \{(q, a, q_{\mathbf{f}}) \mid q \in Q^{\bullet}, \neg Sink(q), a \in A_o, q \not\xrightarrow{a}\} \\ \cup\ & \{(q, a, q_{\mathbf{i}}) \mid q \in Q^{\circ}, \neg Sink(q), a \in A, q \not\xrightarrow{a}\} \ . \end{aligned}$$

Definition 8. *The test case for a BDDTS $\mathcal{B}$ with initialisation $ini \in \mathcal{T}(\emptyset)^{LV}$ such that $[\![IG]\!]_{ini} \equiv \mathbf{true}$ is defined as:*

$$TC(\mathcal{B}, ini) \ = \ TC([\![CRSS(\mathcal{B}, ini)]\!]) \ .$$

For instance, the test case for $\mathcal{B}_2$ is shown in Figure 5.

5 Sequential Composition

We now come to the second main contribution of this paper: the definition of sequential composition for BDDTSs. Two BDDTSs $\mathcal{B}_1, \mathcal{B}_2$ can be sequentially composed wherever the output guard of an open goal location gl of $\mathcal{B}_1$ implies the input guard of $\mathcal{B}_2$. We call such a location *gl composable*. In fact, every composable goal location gl will be 'connected' to the initial location il_2 of $\mathcal{B}_2$. The connection is constructed by defining for each initial switch $(il_2, \alpha, \phi, a, l_2)$ of $\mathcal{B}_2$ a new switch $(gl, \alpha, \phi, a, l_2)$ starting in gl, with the same label and target location. (In this way, il_2 and thus switches $(il_2, \alpha, \phi, a, l_2)$ may become unreachable, but that does not affect the well-definedness of the sequential composition.)

For sequential composition to be well-defined, we require $\mathcal{B}_1$ and $\mathcal{B}_2$ to be *disjoint*, in the sense that *(i)* their locations are disjoint, i.e., $L_1 \cap L_2 = \emptyset$ (which can be assumed without loss of generality because location identities are irrelevant for the semantics); and *(ii)* $\mathcal{B}_1$'s model variables are disjoint from $\mathcal{B}_2$'s context variables and vice versa, i.e., $MV_1 \cap CV_2 = MV_2 \cap CV_1 = \emptyset$. For shared context and model variables, the sequential composition assumes that they are indeed the same. Note that an assignment to a model variable $v \in MV_1 \cap MV_2$ in the $\mathcal{B}_1$-part of the composition may affect the truth of guards in $\mathcal{B}_2$.

Furthermore, we remove the input guard of $\mathcal{B}_2$, because it is guaranteed by the composition, but keep the output guards of $\mathcal{B}_1$'s composable goal locations, because they remain relevant for testing, i.e. for a check of the *Then* step.

Definition 9. *Let $\mathcal{B}_1, \mathcal{B}_2$ be disjoint BDDTSs. The composable goal locations of $\mathcal{B}_1$ are defined as $L_1^c = \{gl \in L_1^\circ \mid OG_1 \downarrow gl \ \wedge \ OG_1(gl) \Rightarrow IG_2\}$. If $L_1^c \neq \emptyset$, the sequential composition $\mathcal{B}_1 \triangleright \mathcal{B}_2$ is defined as $\langle L_1 \cup L_2, V, G_1 \cup G_2, \rightarrow, il_1, IG_1, OG \rangle$, where:*

- *The $\overset{\circ}{\bullet}$-nature of locations is preserved;*
- *$V = IV \cup MV \cup CV$ where each X in IV, MV, CV is defined as $X_1 \cup X_2$;*
- *$\rightarrow \ = \rightarrow_1 \cup \rightarrow_2 \cup \{(l_1, \alpha, \phi, a, l_2) \mid l_1 \in L_1^c, il_2 \xrightarrow{\alpha, \phi, a} l_2\}$;*
- *$OG : l \mapsto \begin{cases} OG_1(l) \wedge OG_2(il_2) & \text{if } l \in L_1^c \text{ and } OG_2 \downarrow il_2 \\ OG_1(l) & \text{if } OG_1 \downarrow l, \text{ and } l \notin L_1^c \text{ or } OG_2 \uparrow il_2 \\ OG_2(l) & \text{if } OG_2 \downarrow l \end{cases}$*

Example 2. We apply the sequential composition on the BDDTS of **Scenario 2** in Figure 2a and the BDDTS of **Scenario 3** in (Figure 2b). The result of the sequential composition is shown in Figure 6. The two BDDTSs can be sequentially composed because the output guard of $\mathcal{B}_2$ at location 2:

$$OG_2(2) = \mathsf{basestation13.overlaymode} = \mathsf{Maintenance} \\ \wedge \mathsf{basestation13.sysmode} = \mathsf{Maintenance}$$

logically implies (here even identical to) the Input Guard of $\mathcal{B}_3$. A new switch (equal to the switch from the initial location of $\mathcal{B}_3$) is added from location 2 of $\mathcal{B}_2$, shown with bold lines in green, and the Input guard of $\mathcal{B}_3$ is removed.

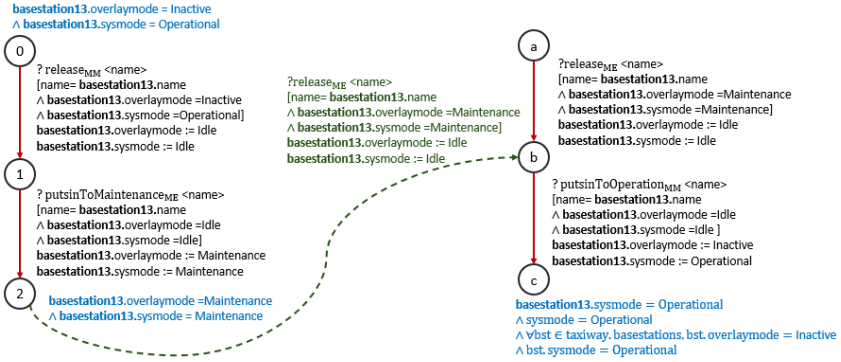


Fig. 6: Sequential Composition $\mathcal{B}_2 \triangleright \mathcal{B}_3$

We note we could slightly adapt the sequential operator to compose two scenarios in a loop if the *Then*-step of each scenario implies the *Given*-step of the other scenario. Looping behavior may well be possible in a software system, while the format of BDD scenarios does not stimulate describing this.

Correctness. We now discuss and show under what circumstances the composed BDDTS $\mathcal{B}_1 \triangleright \mathcal{B}_2$ can replace the original $\mathcal{B}_1$ and $\mathcal{B}_2$ in testing.

Theorem 1 states that if a test derived from the composed system fails on a SuT, the individual systems also generate at least one failing test. In practice, the composed system is capable of finding even more bugs because the model can generate more, various traces due to the different orders the scenarios can be composed, which eventually tests the scenarios with more valuations.

Theorem 2 states the inverse property, namely that any bug found by a test derived from the individual BDDTSs $\mathcal{B}_1$ and $\mathcal{B}_2$ can also be found based on their sequential composition $\mathcal{B}_1 \triangleright \mathcal{B}_2$. This holds when the output guard of the composable locations of $\mathcal{B}_1$ is semantically equivalent to the input guard of $\mathcal{B}_2$ — which is not always the case. In fact, when the output guard is strictly stronger than the input guard, this may indicate a missing (unwritten) scenario in the set. To address this, either a new scenario could be added to cover the part of $\mathcal{B}_2$'s input guard that $\mathcal{B}_1$ did not consider, or $\mathcal{B}_2$ should be re-tested in isolation, focusing on the uncovered part in the given.

However, this is generally not an issue in practice. This is because, in most cases, tests are generated and executed with a single arbitrary or special initialization, and if the test passes, it is already a positive sign. By sequentially composing the scenarios, we not only eliminate the need for repeated initialization phase for each scenario but also, if multiple output guards make the input guard of $\mathcal{B}_2$ true, we test more variations with no additional effort.

Preliminaries. The theorems below crucially rely on the fact that the "goal traces"(traces leading to a goal location having an output guard) of a sequentially composed BDDTS $\mathcal{B}_1 \triangleright \mathcal{B}_2$ are fully characterized by those of $\mathcal{B}_1$ and $\mathcal{B}_2$,

in the following way. For any BDDTS $\mathcal{B}$ and $L \subseteq L_{\mathcal{B}}$, let

$$\Sigma_L(\mathcal{B}) = \{(\sigma, OG(gl)) \mid il_i \xrightarrow{\sigma} gl \in L, OG \downarrow gl\}$$

be pairs of traces of $\mathcal{B}$ to goal locations from L, together with the corresponding output guards; and let $\Sigma(\mathcal{B}) = \Sigma_{L_{\mathcal{B}}}(\mathcal{B})$. It can be shown that $\Sigma(\mathcal{B}_1 \rhd \mathcal{B}_2) = \Sigma(\mathcal{B}_1) \cup \Sigma_{1\cdot 2}$, where

$$\Sigma_{1\cdot 2} = \{(\sigma_1\sigma_2, \phi_2) \mid (\sigma_1, \phi_1) \in \Sigma_{L_1^c}(\mathcal{B}_1), \phi_1 \Rightarrow IG_2, (\sigma_2, \phi_2) \in \Sigma(\mathcal{B}_2)\}.$$

(recalling that L_1^c is the set of composable locations of $\mathcal{B}_1$, see Definition 9).

Theorem 1 (soundness). *Let $\mathcal{B}_1, \mathcal{B}_2$ be sequentially composable BDDTSs, and let $\mathcal{B} = \mathcal{B}_1 \rhd \mathcal{B}_2$. If $TC(\mathcal{B}, ini) \xrightarrow{\sigma} q_{\mathbf{f}}$, then either of the following holds:*

- *$TC(\mathcal{B}_1, ini) \xrightarrow{\sigma} q_{\mathbf{f}}$;*
- *$\sigma = \sigma_1\sigma_2$ for some σ_1, σ_2 with $TC(\mathcal{B}_1, ini) \xrightarrow{\sigma_1} (l, \vartheta)$ and $TC(\mathcal{B}_2, ini_2) \xrightarrow{\sigma_2} q_{\mathbf{f}}$ with $l \in L_1^c$ and $[\![ini_2]\!] = \vartheta{\restriction_{LV_2}}$.*

Theorem 2 (completeness). *Let $\mathcal{B}_1, \mathcal{B}_2$ be sequentially composable BDDTSs, and let $\mathcal{B} = \mathcal{B}_1 \rhd \mathcal{B}_2$ and 0.*

- *If $TC(\mathcal{B}_1, ini_1) \xrightarrow{\sigma} q_{\mathbf{f}}$ then $TC(\mathcal{B}, ini) \xrightarrow{\sigma} q_{\mathbf{f}}$;*
- *If $TC(\mathcal{B}_1, ini_1) \xrightarrow{\sigma_1} (l, \vartheta)$ with $l \in L_1^c$ and $TC(\mathcal{B}_2, ini_2) \xrightarrow{\sigma_2} q_{\mathbf{f}}$ with $\vartheta{\restriction_{LV_2}} = [\![ini_2]\!]$, then $TC(\mathcal{B}, ini) \xrightarrow{\sigma_1\sigma_2} q_{\mathbf{f}}$.*

6 Case Study: Results

Our tool: *RobotFrameworkMBT* is an extension to the Robot Framework(RF) [13], inspired by the theory presented, to combine the BDD approach with MBT. Robot Framework is an open-source automation framework for test and robotic process automation. It uses keyword-driven testing to separate test design from execution, defining actions like clicks or keystrokes. RF supports BDD scenarios in the *Given-When-Then* style, implementing each step with Robot keywords.

The theory and tool correspond with each other as follows. The keywords in RF represent the interactions of BDDTS switches. Guards, assignments, and variable initialization are documented as *model info* in RobotFrameworkMBT. Model info, written in Python syntax, includes :IN: and :OUT: expressions containing boolean terms and assignments. The tool can sequentially compose scenarios by analyzing :IN: and :OUT: expressions in the *Given* and *Then* steps. By applying sequential composition multiple times, it obtains tests with loops. In one respect, the tool implementation deviates from the formal constructions presented in this paper, namely concerning the moment at which variables are initialized. Our formal definition of sequential composition is based on BDDTSs, in particular their output and input guards; variable initialization occurs afterwards, in the transformation to STSs (Definition 3). In contrast, for practical reasons, RobotFrameworkMBT first initializes the variables, before checking the

guards for composition. Though this is less modular and has to be repeated for every choice of variable initialization, this variation potentially gives rise to more connecting states. The validity of the method is not affected.

Below we show the *model info* for the Given step of **Scenario 1** (with corresponding BDDTS in Figure 1). We instantiate the ${AGL function} variable with values 'runway' and 'taxiway'.

```
    [Documentation]    *model info*
...    :IN: system.mode==operational |
...         all([bs.overlay_mode == 'inactive' and
                 bs.system_mode == 'operational' for bs in ${AGL function}.basestations])
```

We applied RobotMBT on the case study as introduced in section 2, using 21 BDD scenarios, including the four as shown in the paper. We categorize the test results we obtained into three groups:

Testing various non-trivial traces. Figure 7a shows test execution after the tool generated the abstract test cases (which are the sequences of BDD scenarios each representing an actual test case), while Figure 7b demonstrates a different order based on sequential composition. Without model info, tests follow the stated order. However, **Scenario 1** and **Scenario 2** cannot be executed in the given order, and require a setup when using RF without RobotMBT, to reach the system state as specified in Given. This is the overhead we noted in section 1.

Testing non-trivial traces of a system is important because it allows us to evaluate the system's behavior under complex, real-world scenarios that go beyond simple or expected use cases. The error in Figure 8 highlights a real scenario where, after the Maintenance Engineer replaces a basestation, it becomes unreachable and is handed back to the Maintenance Manager for operation. The failure occurred because the step implementations did not allow this, though the initial model(the BDDTS as specified by the model info) did. Stakeholders agreed the model should not permit this hand-back.

Finding issues in test implementations. Model-based testing revealed incomplete step implementations that caused false failures when the execution order changed. In Figure 9, the *Given* step should check basestation.sys_mode $\neq$ Operational but it only checked the more specific constraint basestation.sys_mode

(a) trace 1 (b) trace 2

Fig. 7: RF Test Report for Different Execution Orders

= Maintenance, while there is a third possible sys_mode, namely Idle. Hence this case showed that our approach can expose issues in step implementations.

Uncovering difficult-to-find bugs in the system Although we found no system bugs for the scenarios of the partial maintenance feature, we challenged ourselves to automatically trigger and detect a previously resolved issue. This issue was missed by automated tests and only discovered later through manual testing. It was particularly hard to detect as it required both repetition and composition of scenarios. As we stated in section 2, the partial maintenance feature aims to keep the system operational during basestation maintenance. The known issue occurred under the following conditions: a) The system is in partial maintenance, meaning that at least one basestation is in maintenance mode. b) The Air Traffic Controller does repetitive mode switching like putting the system into Idle and then making it Operational again. c) And finally when the Air Traffic Controller switches the lamp levels, the lamp levels do not switch correctly.

To replicate the error, we first created BDD scenarios for system mode and lamp level switching, as these were missing. We then modeled the new scenarios and ran them with **Scenario 1**. The error appeared after the third sequential composition the tool tried, revealing the trace that caused it. While we specifically targeted **Scenario 1**, testing earlier with MBT could have uncovered the issue at some point when running the mode-switching feature with partial maintenance feature, as several scenarios in the suite activate partial maintenance. You can see the test report in Figure 10.

7 Related Work

Composition of BDD Scenarios. Several papers propose the composition of BDD scenarios. In [10], the authors argue that BDD scenarios alone are unsuitable for Cyber-Physical Production Systems due to their complexity. They address this by modeling scenarios with a variant of state machines and composing those with shared events but different pre/post conditions. Unlike their CPPS-specific, manual approach, ours is more general, formally defined, and automated.

Kang et.al provide a translation from BDD scenarios to Timed-Automata (TA) [9]. They formally verify the compatibility of BDD Scenarios and the TA models using UPAAL. With their verification technique, they can also point out

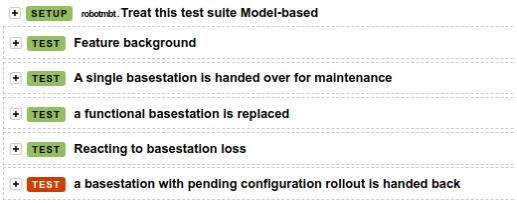


Fig. 8: An example of a non-trivial trace

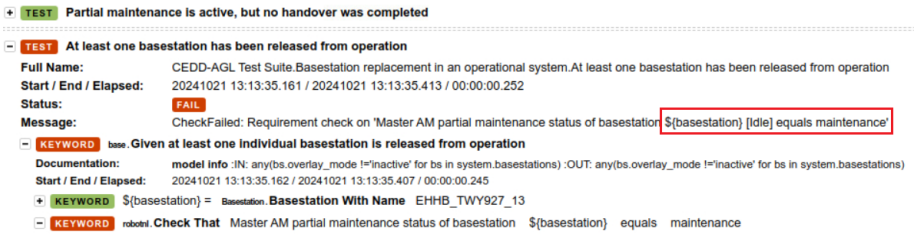


Fig. 9: An example of a test implementation issue

the missing behavior in BDD Scenarios. They manually compose some scenarios with shared actions, but no specific composition operator is introduced and one important missing aspect in their work is that output actions are not considered.

Silva introduces a DSL to identify the scenario entities [14]. The DSL contains an abstract Given-When-Then structure. In each step of the abstract scenario, a refined scenario can be written that connects the related scenarios in this abstract way. We currently do not have a formalism for refinement but a prototype implementation of refinement is available in our RobotFrameworkMBT tool. With the tool SkyFire, they generate the BDD scenarios from UML diagrams [11]. The generated scenarios are long, and connect scenarios sequentially, as the UML diagrams specify. They generate tests using Cucumber. Their approach is opposite to ours, we model scenarios while they write scenarios from models.

Composition Operators for Transition Systems. Several papers have proposed composition operators for Labeled Transition Systems (LTSs). Parallel Composition of LTSs in ioco-theory is defined in [2] and is further improved by formally defining mutual acceptance in [6]. Conjunction, parallel composition, and Quotient are also defined in [1] and the parallel composition is revised in [7]. In [8] a conjunction operator is defined on LTSs with inputs and outputs, such that conjunction expresses the intersection over outputs. Our composition operators for

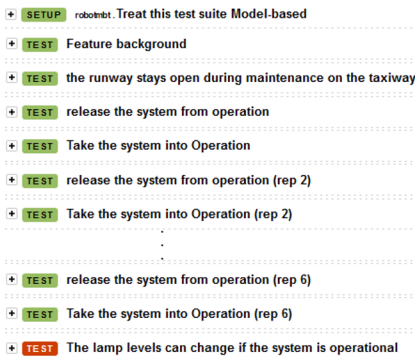


Fig. 10: An example of a known system bug: Mode Switching Error

sequentially composing BDD scenarios are defined directly on BDDTS (which is a variant of STS), making them suitable for BDD scenarios.

8 Conclusion and Future Work

This paper presents a compositional approach to Behavior-Driven Development (BDD) where system behavior is specified and tested through scenarios written in natural language. Specifically, we introduced a sequential composition operator for BDD Transition Systems (BDDTS), which are formal models for BDD scenarios. By composing BDDTSs, we can generate more and longer test cases from composed models that account for various integrations of behaviors. These integrations might be challenging to address when writing and testing individual BDD scenarios. The composition also aids in identifying missing scenarios that might be overlooked when only describing small features in single scenarios.

In future work we plan to introduce another composition operator, namely "disjunction", to connect scenarios that share actions or similar pre-conditions. We will evaluate disjunction composition in an industrial case study. We should also investigate how to deal with quiescence [16]. In addition, we could investigate NLP techniques to facilitate the translation between BDD Scenarios and BDDTS or BDD scenarios to some intermediate BDD language [18]. Finally, we would like to experiment with using MBT techniques to generate tests with diverse data values, and how that improves bug detection compared to the standard BDD approach where a BDD scenario has just one test with fixed data values.

Acknowledgement. This publication is part of the project *TiCToC -Testing in Times of Continuous Change-* with project number 17936 of the research program *MasCot-Mastering Complexity-* which is supported by NWO.

References

1. Beneš, N., Daca, P., Henzinger, T.A., Křetínský, J., Ničković, D.: Complete composition operators for ioco-testing theory. In: Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering. p. 101–110. CBSE '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737166.2737175
2. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with ioco. In: Petrenko, A., Ulrich, A. (eds.) Formal Approaches to Software Testing. pp. 86–100. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
3. Van den Bos, P., Tretmans, J.: Coverage-based testing with symbolic transition systems. In: Beyer, D., Keller, C. (eds.) Tests and Proofs. pp. 64–82. Springer International Publishing, Cham (2019)
4. Chelimsky, D., Astels, D., Helmkamp, B., North, D., Dennis, Z., Hellesoy, A.: The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends. Pragmatic Bookshelf, 1st edn. (2010), `https://dl.acm.org/doi/10.5555/1965448`

5. Cucumber, https://cucumber.io/docs/guides/overview/
6. van Cuyck, G., van Arragon, L., Tretmans, J.: Compositionality in model-based testing. In: Bonfanti, S., Gargantini, A., Salvaneschi, P. (eds.) Testing Software and Systems. pp. 202–218. Springer Nature Switzerland, Cham (2023)
7. Daca, P., Henzinger, T.A., Krenn, W., Nickovic, D.: Compositional specifications for ioco testing. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 373–382 (2014). https://doi.org/10.1109/ICST.2014.50
8. Janssen, R.: Combining partial specifications using alternating interface automata. In: Wehrheim, H., Cabot, J. (eds.) Fundamental Approaches to Software Engineering. pp. 462–481. Springer International Publishing, Cham (2020)
9. Kang, E.Y., Silva, T.R.: Towards formal verification of behaviour-driven development scenarios using timed automata. In: 2023 30th Asia-Pacific Software Engineering Conference (APSEC). pp. 612–616 (2023). https://doi.org/10.1109/APSEC60848.2023.00081
10. Kannengiesser, U., Krenn, F., Stary, C.: A behaviour-driven development approach for cyber-physical production systems. In: 2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS). vol. 1, pp. 179–184 (2020). https://doi.org/10.1109/ICPS48405.2020.9274755
11. Li, N., Escalona, A., Kamal, T.: Skyfire: Model-based testing with cucumber. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 393–400 (April 2016). https://doi.org/10.1109/ICST.2016.41
12. Nagy, G., Rose, S.: The BDD Books – Formulation Document examples with Given/When/Then. Leanpub (2021), https://leanpub.com/bddbooks-formulation
13. Robot framework, https://robotframework.org/
14. Silva, T.R.: Towards a domain-specific language for behaviour-driven development. In: 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 283–286 (2023). https://doi.org/10.1109/VL-HCC57772.2023.00054
15. Specflow, https://docs.specflow.org/en/latest/
16. Tretmans, J.: Model-Based Testing with Labelled Transition Systems, pp. 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_1
17. Zameni, T.: Executable bdd scenario models for robotframeworkmbt. https://doi.org/10.5281/zenodo.14892704
18. Zameni, T., van den Bos, P., Rensink, A., Tretmans, J.: An intermediate language to integrate behavior-driven development scenarios and model-based testing. In: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C). pp. 199–206 (2024). https://doi.org/10.1109/SANER-C62648.2024.00033
19. Zameni, T., van Den Bos, P., Tretmans, J., Foederer, J., Rensink, A.: From bdd scenarios to test case generation. In: 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 36–44 (2023). https://doi.org/10.1109/ICSTW58534.2023.00019