JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Disjunction Composition of BDD Transition Systems for Model-Based Testing

Tannaz Zameni, Petra van den Bos, and Arend Rensink
University of Twente, The Netherlands

ABSTRACT We introduce a compositional approach to model-based test generation in Behavior-Driven Development (BDD). BDD is an agile methodology in which system behavior is specified through textual scenarios that, in our approach, are translated into transition systems used for model-based testing. This paper formally defines *disjunction composition*, to combine BDD transition systems that represent alternative system behaviors. Disjunction composition allows for modeling and testing the integrated behavior while ensuring that the testing power of the original set of scenarios is preserved. This is proved using a symbolic semantics for BDD transition systems, with the property that the symbolic equivalence of two BDD transition systems guarantees that they fail the same test cases. Also, we demonstrate the potential of disjunction composition by applying the composition in an industrial case study.

KEYWORDS BDD Transition Systems, Symbolic Transition Systems, Disjunction Composition, Behavior-Driven Development, Model-Based Testing.

1. Introduction

Testing often is the primary method for verifying system behavior and is a critical part of software development. But as systems grow in complexity, ensuring the correctness of all integrated behaviors becomes increasingly challenging. Model-based testing (MBT) offers a powerful solution by automating the generation of diverse test cases derived from formal models: precise specifications that define what the system should and should not do. Additionally, using partial models and composing them offers an effective way to ease model construction and obtain larger models that detect bugs that are difficult and expensive to find through manual testing. However, creating and maintaining such formal models can be time-consuming and requires specialized skills, which often makes MBT hard to adopt in everyday software development.

On the other hand, Behavior-driven Development (BDD) is a widely adopted agile methodology that offers a more accessible way to describe system behavior for testing. In BDD, different stakeholders, such as product owners, developers and testers, collaboratively discover system behaviors that are captured in structured documents. A common structure is the *Given-When-Then* format of the Gherkin language (Rose & Nagy 2021; Chelimsky et al. 2010): *Given* a precondition for the required system state, *When* an action is performed on/by the system, *Then* specified actions and resulting state are expected. Tools like Cucumber (Cucumber 2026) and Reqnroll (Reqnroll 2026) enable users to manually implement the steps of BDD scenarios. However, they lack formal semantics: due to use of natural language, BDD scenarios may be ambiguous. Moreover, BDD scenarios lack mechanisms to automatically compose scenarios and capture their integrated behavior.

In this paper, we combine BDD and MBT. We start from BDD Transition Systems (BDDTSs): formal models that can be derived from BDD scenarios (see, e.g., (Zameni et al. 2024, 2023)). This paper introduces *disjunction composition* for BDDTS. It combines BDDTSs that represent alternative valid ways for a system to behave. When applied, it integrates the overlapping, but partly different, behaviors of two BDDTSs into one BDDTS. In particular, if both BDDTS describe the same action in their ***When*** step, the composition takes this action as well. Actions that only appear in the ***When*** step of one of the BDDTS are included in the composition as well. This way, we faithfully model that ***When*** steps describe which actions a system *may* perform. In contrast, ***Then*** steps describe *expected* behavior.

JOT reference format:
Tannaz Zameni, Petra van den Bos, and Arend Rensink. *Disjunction Composition of BDD Transition Systems for Model-Based Testing*. Journal of Object Technology. Vol. 25, No. 3, 2026. http://dx.doi.org/10.5381/jot.2026.25.3.a26

The disjunction composition allows all actions of ***Then*** steps specified by both BDDTS. Other actions are discarded, since the BDDTS disagree on those expectations (and their tests would as well), hence they cannot both be true. (The consequence of discarding an action from the model is that, if a System under Test does perform that action, the test fails.) Hence, disjunction composition not only facilitates the integration of two BDDTS but also reveals modeling inconsistencies.

Concrete advantages of generating tests on the basis of a composed BDDTS, which covers more behavior, are that (i) combinations of behaviors can now be tested, when testing them separately may not reveal a bug that only surfaces in the combination, (ii) individual BDDTSs can be maintained and adapted while the System Under Test evolves while test generation from the composition provides test cases covering a wide range of possibilities for which a variation of test cases can be selected and generated, (iii) the System under Test has to be initialized (i.e., set up) less frequently, since several behaviors can now be covered with a single test, (iv) compositions of BDDTS may reveal modeling inconsistencies and implicit assumptions, (v) the resulting tests more often give a definite verdict (pass or fail) rather than being inconclusive (as happens when the system displays behavior not covered by the model).

The name *disjunction composition* reflects the intuition that any behavior that either of the composed features can perform is allowed in the When steps, reflecting alternative (disjunctive) possibilities. However, some aspects of the construction (such as discarding non-overlapping actions of ***Then*** steps) actually have a more conjunction-like flavor.

This paper makes four main contributions:

1. We formally define disjunction composition;
2. We define a symbolic semantics for BDDTS to reason symbolically about equivalence for testing, and use this to show that the composition of two BDDTS is equivalent to the conjunction of the semantics of those two BDDTS;
3. We prove correspondence between symbolic semantics of a BDDTS and the concrete test cases that can be derived from it;
4. We apply disjunction composition on an industrial case study, where we apply and compare our approach with the Dutch Railway's approach of using BDD scenarios for testing their information boards that display train departures.

Figure 1 shows an overview of the main elements of the setup as envisioned in this paper. The first step, from BDD scenarios to BDDTSs (which involves manual modeling), is actually not in the scope of this paper (we have addressed this previously in (Zameni et al. 2024).) All other steps are formalized in this paper, and can, in principle, be automated. BDDTSs are the central model of this paper (Section 2). The *concrete semantics* of BDDTS (Section 5) on the left of the overview, is based on an interpretation from Symbolic Transition Systems (STSs) (Van den Bos & Tretmans 2019) in terms of Labeled Transition Systems (LTSs) (Tretmans 2008), where transitions are instantiated with concrete data values, similar to (Frantzen et al. 2005). This semantics is well-suited for test case (TC) generation and understanding system behaviors at the instance level. The symbolic semantics (Section 3) describes the system in two ways. First, Execution Conditions (ECs) capture the conditions under which a sequence of transitions can happen. Second, Goal Implications (GIs) express how these conditions lead to expected outcomes, corresponding to the goals defined in the ***Then*** steps of a BDDTS. In other words, ECs tell us when things can happen, and GIs tell us what should be the result if they do happen. Symbolic semantics is based on saturated BDDTS (Section 2), which is a version of the BDD transition system where every possible output and the relevant inputs are explicitly represented. This semantics abstracts away from specific data values, allowing reasoning over sets of behaviors in a compact, generalized form. We use this to reason about equivalence of two BDDTS with their disjunction composition (Section 4). Finally, we discuss the case study (Section 6), related work (Section 7) and conclusions (Section 8). Proofs of the results can be found in the technical report of this paper (Zameni et al. 2026).

2. BDD Transition Systems

Before introducing the formal machinery, we present two abstract scenarios that will be used later to illustrate how the disjunction composition of scenarios works, and in particular, which scenario structures are well-suited to such composition. The scenarios are deliberately schematic: the inputs, outputs, and conditions are kept abstract so that the discussion focuses

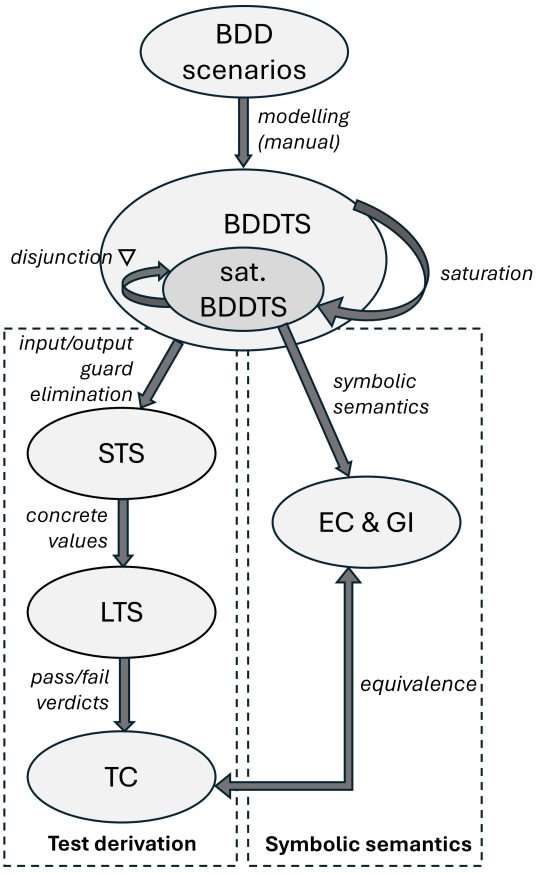


Figure 1 Overview of our paper

on their structural shape rather than on any concrete system. The corresponding formal model is given later in Figure 2.

Scenario 1 High level example Scenario: Output with $x \geq 10$
- ***Given*** the system is in a state where $Y \leq 5$ holds
- ***When*** the user provides input **a**
- ***Then*** the system outputs **b** with x
  And $x \geq 10$
  And the system again outputs **b** with x
  And $x \geq 10$

Scenario 2 High level example Scenario: Output with $x \geq 15$
- ***Given*** the system is in a state where $Y \geq 3$ holds
- ***When*** the user provides input **a**
- ***Then*** the system outputs **b** with x
  And $x \geq 15$
  And the system again outputs **b** with x
  And $x \geq 15$

Both scenarios share the same input **a** and the same output pattern (two successive emissions of **b** with a value x), but differ in their *Given* preconditions on Y, and in the constraints they impose on x. This shared input/output skeleton with different guards and output constraints is a good example of the structural pattern under which disjunction composition is useful. Section 4 returns to these scenarios to make the point precise.

Turning these informal scenarios into objects we can reason about requires a formal model. To define BDDTS formally, we need to introduce several notations, namely for functions, for terms and assignments, and for states and labeled transitions.

Functions. We write a function f from domain X to codomain Y as $f : X \to Y$, and a partial function as $f : X \hookrightarrow Y$. We also write Y^X for the set of all such functions, and $Y_\perp^X$ for the partial functions. With $f \downarrow x$ we denote that f is defined for x, and with $f \uparrow x$ that f is undefined for x. This is extended to $f \downarrow Z$ and $f \uparrow Z$ (with $Z \subseteq X$) to mean (un)definedness for *all* $x \in Z$.

Given two partial functions $f_A : A \hookrightarrow B$ and $f_C : C \hookrightarrow D$ such that $f_A \downarrow x$ and $f_C \downarrow x$ implies $f_A(x) = f_C(x)$ for all $x \in A \cap C$, we define their union $f_A \sqcup f_C : (A \cup C) \hookrightarrow (B \cup D)$ as:

$$f_A \sqcup f_C : x \mapsto \begin{cases} f_A(x) & \text{if } x \in A \text{ and } f_A \downarrow x \\ f_C(x) & \text{if } x \in C \text{ and } f_C \downarrow x. \end{cases}$$

Terms and assignments. In the following, we briefly review common algebraic concepts like terms, variables and assignments, and their semantics, without going into full formal detail.

- We build *terms* from functions and variables (as usual). The set of all variables is denoted $\mathcal{V}$, and the set of terms over variables $V \subseteq \mathcal{V}$ is denoted $\mathcal{T}(V)$. A term t is called *ground* if $t \in \mathcal{T}(\emptyset)$. We assume that terms are well-typed. Terms of type Bool, over variables V, are denoted $\mathcal{T}_{\mathsf{Bool}}(V)$.
- We write $t[x/y]$ with $t \in \mathcal{T}(X)$ and $x, y \in X$ for the substitution of all y by x in t.
- An *assignment* is a partial function $a \in \mathcal{T}(V)_\perp^Y$ for some $V, Y \subseteq \mathcal{V}$. $id_V \in \mathcal{T}(V)^V$ is the *identity assignment* over V, defined as $id(v) = v$ for all $v \in V$. We will also use $a_1 \sqcup a_2$ if a_1 and a_2 are *compatible* ($a_1 \Leftrightarrow a_2$, see below). We apply an assignment a to a term t by writing $t[a]$ (which syntactically substitutes, in t, all x for which $a \downarrow x$ by $a(x)$). This notation is extended pointwise to assignments: $a_1[a_2]$ is the assignment defined by $\{x \mapsto a_1(x)[a_2] \mid a_1 \downarrow x\}$.
- The set of variables actually occurring in a term $t \in \mathcal{T}(V)$ are denoted $vars(t)$. For an assignment a, we use $vars(a) = \bigcup_{a \downarrow x} vars(a(x))$.
- The value of a term $t \in \mathcal{T}(V)$ is generated by a *valuation* $\vartheta : V \to \mathcal{U}$, where $\mathcal{U}$ denotes the domain of (semantic) values (of all types). For $t \in \mathcal{T}(V)$ and $\vartheta \in \mathcal{U}_\perp^V$ with $\vartheta \downarrow vars(t)$ we write $[\![t]\!]_\vartheta \in \mathcal{U}$ to denote the value of t under ϑ, i.e. the value that is obtained by substituting the values for variables according to ϑ in the term t, and then evaluating the result. Likewise, for assignments $a \in \mathcal{T}(V)_\perp^V$, $[\![a]\!]_\vartheta \in \mathcal{U}_\perp^V$ is the valuation that maps each $x \in V$ to $[\![a(x)]\!]_\vartheta$. If t resp. the images of a are ground, we don't need ϑ and will just write $[\![t]\!]$ and $[\![a]\!]$.
- Terms $t_i \in \mathcal{T}(V_i)$ for $i = 1, 2$ are *semantically equivalent*, denoted $t_1 \equiv t_2$, if $[\![t_1]\!]_\vartheta = [\![t_2]\!]_\vartheta$ for all $\vartheta \in \mathcal{U}^{V_1 \cup V_2}$. Assignments $a_i \in \mathcal{T}(V_i)^{Y_i}$ for $i = 1, 2$ are *semantically compatible*, denoted $a_1 \Leftrightarrow a_2$, if $a_1(x) \equiv a_2(x)$ for all $x \in Y_1 \cap Y_2$. If $a_1 \Leftrightarrow a_2$ then $a_1 \sqcup a_2$ is well-defined, as announced above.
- For predicates $t_i \in \mathcal{T}_{\mathsf{Bool}}(V)$ and valuations ϑ with $\vartheta \downarrow vars(t_i)$ for $i = 1, 2$, we use $\vartheta \models t_1$ ("ϑ satisfies t_1") as an alternative notation for $[\![t_1]\!]_\vartheta = \mathbf{true}$, and $t_1 \Rrightarrow t_2$ to denote semantic implication, i.e., $\vartheta \models t_1$ implies $\vartheta \models t_2$ for all $\vartheta \in \mathcal{U}^V$.

In the above, we ignore the choice of the particular functions and predicates from which terms are built. This is an issue treated in more detail in (Zameni et al. 2024), where we introduced the concept of a *Domain-Specific Interpretation*: essentially an algebra that captures relevant information about the domain, such as the available types, values, and operations on them.

States and labeled transitions. 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 such transition relations, we adopt some common notations. To start, $q_0 \xrightarrow{a_0 \cdots a_{n-1}} q_n$ for $n \geq 0$ denotes $(q_i, a_i, q_{i+1}) \in \to$ for $0 \leq i < n$; next, $q \xrightarrow{\sigma} P$ with $P \subseteq Q$ and $\sigma \in A^*$ denotes that there is some $q' \in P$ such that $q \xrightarrow{\sigma} q'$; and finally, $q \xrightarrow{\sigma}$ abbreviates $q \xrightarrow{\sigma} Q$. A state q is called a *sink*, denoted $Sink(q)$, if $\nexists a : q \xrightarrow{a}$. The subset of sink states in $P \subseteq Q$ is denoted P_{Sink}.

A set of labels A is usually partitioned in a set of inputs A_i and outputs A_o, i.e. $A = A_i \cup A_o$, and $A_i \cap A_o = \emptyset$. An output denotes an action of the System under Test, that is provided to, or observed by its environment (e.g., a system user, another system, or the testing tooling). Conversely, an input denotes an action performed by the environment that is provided to the System under Test.

In several cases, the set of states Q will be partitioned into $Q = Q^\circ \cup Q^\bullet$ of *open* and *closed* elements, to record the intended interpretation of "missing outputs", i.e., output actions

for which there is no outgoing transition from a given state. If a system actually performs such a missing output, contrary to what the model specifies, how should this be dealt with? From closed$^\bullet$ states this is considered forbidden (and will lead to a failing test) whereas from open$^\circ$ states this is considered unspecified (and will lead to inconclusive). These different interpretations are inspired by the meaning of actions specified in ***then*** vs. ***when*** steps of a BDD scenario. We call sets Q that are partitioned in this way $^\circ_\bullet$*-natured*. For a $^\circ_\bullet$-natured set Q we will use $\mathsf{N} : Q \to \{\circ, \bullet\}$ to retrieve the nature.

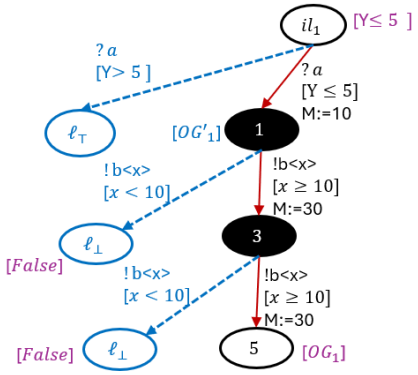
For the case of BDDTSs, defined below, the transitions are defined between *locations* (which we denote L rather than Q as above), and its labels (which we denote Λ rather than A) are structures consisting of several elements, among which are so called *interactions*. The transitions themselves are called *switches* to reflect their more complex structure. An interaction $\alpha = g\,\bar{iv}$ consists of *(i)* a *gate* $g \in \mathcal{G}$, where $\mathcal{G}$ is a universe of gates, partitioned into input gates $\mathcal{G}_i$ and output gates $\mathcal{G}_o$, and *(ii)* a sequence of *interaction variables* $\bar{iv} = iv_1 \cdots iv_n \in IV^*$, where $IV \subseteq \mathcal{V}$ is a special set of variables. A sequence of zero variables is denoted with ϵ. We use g_α and $\bar{iv}_\alpha$ to denote the constituent elements of an interaction α. The universe of interactions $\mathcal{I} \subseteq \mathcal{G} \times IV^*$ forms a one-to-one relation, i.e., gates have fixed interaction variables. For a subset $G \subseteq \mathcal{G}$, we use $\mathcal{I}(G) = \{\alpha \in \mathcal{I} \mid g_\alpha \in G\}$ to denote the subset of interactions with gates in G. In terms of the general setup discussed above, the partitioning of Λ into input labels Λ_i and output labels Λ_o is determined by the input/output-nature of the interaction gate contained in the label.

BDDTS. From these basic elements we now define the core notion of a *BDD Transition System* (BDDTS), inspired by (Zameni et al. 2024), which formally captures the behavior prescribed by a single BDD scenario or (alternatively) the composition of several BDD scenarios. Figure 2 shows two example BDDTSs of **Scenario 1** and **Scenario 2** that we will compose later.

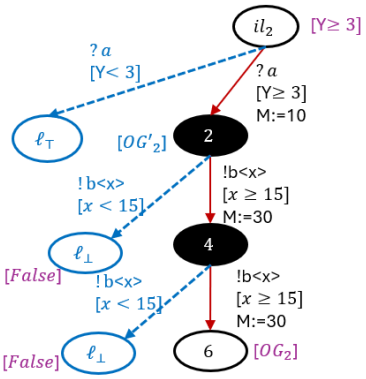
Definition 1 (BDD Transition System) *A* BDD Transition System *(BDDTS) is a tuple* $\mathcal{B} = \langle L, V, G, \to, il, IG, OG\rangle$*, where*

- L *is a* $^\circ_\bullet$*-natured set of* locations*;*
- V *is a set of* location variables*, disjoint from IV and partitioned into MV (model variables) and CV (context variables);*
- $G \subseteq \mathcal{G}$ *is a set of* gates*. We write* $G_i = G \cap \mathcal{G}_i$ *and* $G_o = G \cap \mathcal{G}_o$*, and use IV to denote the set of interaction variables of G (and hence of* $\mathcal{B}$*).*
- $\to \subseteq L \times \Lambda \times L$ *is a* switch relation*, with label set* $\Lambda = \mathcal{I}(G) \times \mathcal{T}_{\mathsf{Bool}}(V \cup IV) \times \mathcal{T}(V \cup IV)^{MV}_{\perp}$*;*
- $il \in L^\circ$ *is the* initial location *(which is open);*
- $IG \in \mathcal{T}_{\mathsf{Bool}}(V)$ *is the* input guard *(which can also be thought of an* initial guard*);*
- $OG : L \hookrightarrow \mathcal{T}_{\mathsf{Bool}}(V)$ *is a partial map from locations to* output guards*.*

$\mathcal{B}$ *is* deterministic*, meaning that: if* $t_i = (sl, \alpha, \phi_i, a_i, tl_i) \in \to$ *for* $i = 1, 2$*, then either* $t_1 = t_2$ *or* $\phi_1 \wedge \phi_2 \equiv \mathbf{false}$*. The* active variables $V_l \subseteq V$ *of a location* $l \in L$ *are recursively defined as the smallest set such that* $vars(OG(l)) \subseteq V_l \cup CV$ *if* $OG \downarrow l$*, and* $vars(\phi_t) \cup vars(a_t) \cup V_{tl_t} \subseteq V_{sl_t} \cup CV \cup IV$ *for all* $t \in \to$*. For all* $t \in \to$ *we require* $a_t \downarrow V_{tl_t}$*: all active target location variables must be assigned.*



(a) BDDTS B_1.



(b) BDDTS B_2.

Figure 2 Two BDDTSs with matching gates but different guards. The solid, red switches are those originally present in the BDDTS, while the dashed, blue switches denote switches that are added by saturation (Definition 2). Input gate a? and output b! are used on the switches where gate b! has interaction variable x. The guards of switches, the input guard of the initial location and the output guards of other locations are stated between '[' and ']'. Assignments use ':=' notation.

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. When considering multiple BDDTSs $\mathcal{B}_i$, we assume they are *compatible*, i.e., share the same G_i and V_i. Hence, we write simply G and V. The components of a switch $t \in \to$ are denoted sl_t (the *source location*), $\alpha_t = (g_t, \bar{iv}_t)$ (the *interaction*), ϕ_t (the *switch guard*, a boolean term), a_t (the *assignment*, a partial function from MV to terms over the active variables of the target location) and tl_t (the *target location*); for a switch denoted t_i, we write sl_i, α_i etc. We call a location l a *goal location* if $OG \downarrow l$.

An important aspect is the distinction between model variables and context variables. Context variables are assumed to be directly observable or read from the System Under Test (SuT); they reflect the actual runtime state or outputs of the system. Model variables, in contrast, belong to the formal model and represent expected or hypothesized state. As a consequence,

model variables can be changed through assignments in BDDTS switches (such an assignment reflects an expectation of how the system state changes), in contrast to context variables.

A BDDTS does contain an initialization of its model variables or context variables; however, the Input Guard typically constrains their expected initial value (possibly even to a particular constant). Upon instantiation of a BDDTS (in our setting, when it is transformed to an STS and from there to a Test Case), initial values have to be provided that satisfy the Input Guard. When a context variable occurs in a switch guard, its value is taken to be unchanged with respect to the initial state; when it occurs in an output guard, however, it is compared with the actual runtime state at that moment. (How this comparison is performed is fixed in the transformation to an STS.)

While model variables are not inherently tied to the SuT, they may be compared against context variables — e.g., in guards or assertions — to express that the observed system behavior conforms to the model's expectations.

Example 2.1 *Figure 3b shows the BDDTS for **Scenario 3** of Figure 3a. We note that the correspondence by a BDD scenario and its BDDTS is discussed in previous work (Zameni et al. 2024). The **Given** step is represented by the input guard on location 0. The When step is represented by the* $!\mathsf{verify}_{\mathsf{badge}}$ *output switch, and the first part of **Then**, i.e. before **And**, is represented by the* $!\mathsf{trigger}_{\mathsf{door}}$ *output switch. Observe that such a **Then** switch starts from a closed location, while the switch resulting from a **When** step starts from an open location. The second part of the Then step, i.e. after **And**, is formalized as the output guard of location 2.*

Model variable $\mathsf{A}_{\mathsf{badge}}$ *contains the list of all authorized badges, while* $\mathsf{P}_{\mathsf{badge}}$ *is a context variable of the badge provided in the **Given** step.* $\mathsf{P}_{\mathsf{badge}}$ *is a context variable as it must be read from the system, while* $\mathsf{A}_{\mathsf{badge}}$ *is a list stored in the model. With model variable* $\mathsf{AccessGranted}$ *we memorize that access was verified in the* $!\mathsf{verify}_{\mathsf{badge}}$ *switch, so that this can be used to guard the* $!\mathsf{trigger}_{\mathsf{door}}$ *switch. The* Door *context variable provides the final status of the door, OPEN or CLOSED.*

BDDTSs were originally introduced as a basis for deriving test cases. In such a test case, the interpretation of a concrete system action that fails to satisfy any of the switch guards in a given location l (in other words, a system action that is "absent" from the BDDTS) depends on the $\overset{\circ}{\bullet}$-nature of l: for open states and for input actions, the test is inconclusive, whereas for output actions from closed states, the test fails. Instead of directly going from BDDTSs to test cases, however, we first compose several (similar) BDDTSs, resulting in a more comprehensive model that combines their testing power. (This is the core contribution of this paper.) For the composition to work as required, we need to first *saturate* the BDDTS by adding switches to capture the intended interpretation of "absent" system actions.

Definition 2 (saturated BDDTS) *A BDDTS $\mathcal{B}$ is called* saturated *if:*

1. *For any $l \in L_{\mathcal{B}}$ and $\alpha \in \mathcal{I}(G)$, the set $\{t_1, \ldots, t_n\} \subseteq \rightarrow_{\mathcal{B}}$ of all switches with $sl_i = l$ and $\alpha_i = \alpha$ is either empty or satisfies $\bigvee \phi_i \equiv \mathbf{true}$;*
2. *For any $l \in L^{\bullet}_{\mathcal{B}}$ and $\alpha \in \mathcal{I}(G_o)$, there is a switch $l \xrightarrow{\alpha,\phi,a}_{\mathcal{B}}$ for some guard ϕ and assignment a.*
3. *For any $il_{\mathcal{B}} \xrightarrow{\alpha,\phi,a}_{\mathcal{B}} l$, either $\phi \Rrightarrow IG$ or $l \in L^{\circ}_{\mathcal{B},Sink}$ and $OG \uparrow l$.*

Clause 1 requires that, for every location and interaction α, either there is no outgoing α-switch at all or there is a switch for all α-based interaction values. Clause 2 requires that, for every *closed* location and *output* interaction α, there is an outgoing α-switch (so the first case of Clause 1 does not occur). Clause 3, finally, encapsulates the intuition that initial switches that correspond to behavior disallowed by the input guard always give rise to an inconclusive verdict. This is achieved by insisting that such switches immediately end in an open, non-goal sink location.

Turning an arbitrary BDDTS into a saturated one (i.e., *saturating* it) consists of three steps:

1. **Input guard propagation**: Since a BDDTS only applies to systems that satisfy its input guard, we make this explicit by adding the input guard to the guards of all existing initial switches.
2. **Completion for existing interactions**: For all sets of switches T from a certain location l with the same interaction α, we add a *saturating switch* from l for α, with a guard equivalent to the negation of all T-guards. Consequently, after saturation, location l has a switch whose guard evaluates to true for any valuation of variables. Saturating switches from open locations lead to a state where any action is allowed ($\ell_\top$ in the definition below), whereas those from closed locations lead to a state ($\ell_\bot$) which has output guard **false** and hence is essentially a failure state.
3. **Completion for absent output interactions**: If l in the above is a closed location, we also add switches for output gates that are not enabled, again to $\ell_\bot$, because these switches represent disallowed behavior.

Definition 3 (saturating a BDDTS) *Given a BDDTS $\mathcal{B} = \langle L, V, G, \rightarrow, il, IG, OG\rangle$, its saturation is defined as*

$$\mathcal{B}^{sat} = \langle L \cup \{\ell_\top, \ell_\bot\}, V, G, \rightarrow_{sat}, il, IG, OG^{sat}\rangle$$

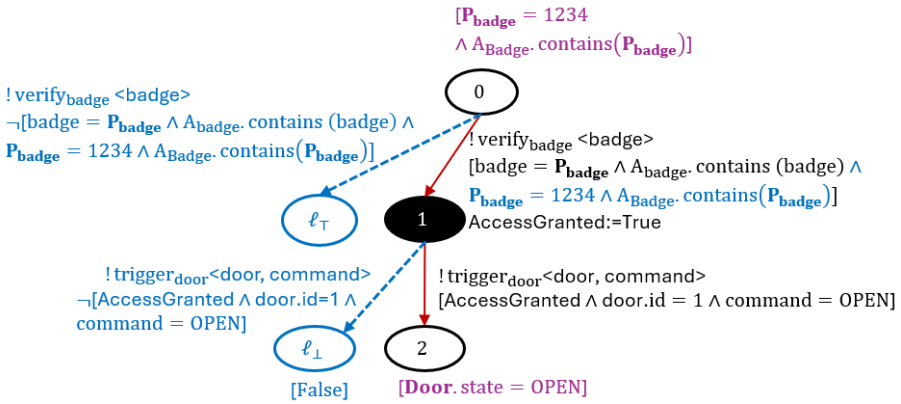
Here $\ell_\top, \ell_\bot$ are fresh open locations such that $OG^{sat} = OG \sqcup \{(\ell_\bot, \mathbf{false})\}$ (hence $OG^{sat} \uparrow \ell_\top$). Switch relation $\rightarrow_{sat}$ is constructed from IG-enriched switches $\rightarrow_{IG}$ and saturating switches $\rightarrow_\top$, $\rightarrow_\bot$ to (respectively) $\ell_\top$ and $\ell_\bot$, using all enabled guards $\Phi_{l,\alpha}$ from a location l for interaction α, as follows:

$$\begin{aligned}
\rightarrow_{IG} &= \{t \in \rightarrow \mid sl_t \neq il\} \\
&\quad \cup \{(il, \alpha_t, IG \wedge \phi_t, a_t, tl_t) \mid t \in \rightarrow, sl_t = il\} \\
\Phi_{l,\alpha} &= \{\phi_t \mid t \in \rightarrow_{IG}, sl_t = l, \alpha_t = \alpha\} \\
\rightarrow_\top &= \{(l, \alpha, \neg \bigvee \Phi_{l,\alpha}, \epsilon, \ell_\top) \mid (l \in L^{\circ} \vee g_\alpha \in G_i), l \xrightarrow{\alpha,\phi,a}\} \\
\rightarrow_\bot &= \{(l, \alpha, \neg \bigvee \Phi_{l,\alpha}, \epsilon, \ell_\bot) \mid l \in L^{\bullet}, g_\alpha \in G_o\} \\
\rightarrow_{sat} &= \rightarrow_{IG} \cup \rightarrow_\top \cup \rightarrow_\bot
\end{aligned}$$

Note that, as a special case, $\rightarrow_\bot$ contains $(l, \alpha, \mathbf{true}, \epsilon, \ell_\bot)$ for all $l \in L^{\bullet}$ and $\alpha \in \mathcal{I}(G_o)$ such that $\nexists t \in \rightarrow : sl_t = l \wedge \alpha_t = \alpha$. $\mathcal{B}^{sat}$ is deterministic as required by Definition 1, and the following proposition holds by construction.

Scenario 3 Door access with badge
- ***Given*** the badge ID 1234 is authorized,
  And badge ID 1234 is presented at the door,
- ***When*** the system verifies the badge,
- ***Then*** the system sends the command to open the door
  And the state of the door is now changed to open.

(a) Scenario 3 describes the verification of a badge to get access through a door.



(b) BDDTS $\mathcal{B}$ has open locations 0 and 2, with input and output guard resp., closed location 1, and the two red switches. The saturation of $\mathcal{B}$ also includes the dashed switches and blue expressions.

Figure 3 A BDD scenario (a) and its corresponding BDDTS (b)

Proposition 1 *For any BDDTS $\mathcal{B}$, $\mathcal{B}^{sat}$ is saturated.*

In Figure 3b, two dashed blue switches are added through saturation. Also, the input guard is conjoined, in blue, to the guard of the red switch from location 0.

3. Semantics of BDDTS in terms of Goal Implications

The essence of saturated BDDTSs is captured by the traces (i.e., sequences of interactions) leading to *goal locations*. We capture the semantics of such traces by collecting the successive switch conditions into a so-called *path condition* — similar to the path conditions for STSs in (Van den Bos & Tretmans 2019). To define this formally, we need some more notation.

- To refer to the value of a variable x at some point in the past, we will use $x^{(i)}$ with $i \in \mathbb{N}$. To be precise, we assume that for all x and all $i > 0$, $x^{(i)}$ is a distinct (fresh) variable referring to the value of x at i time instances in the past. $x^{(0)}$ is the same as x. In addition, $x{\Uparrow}$ *up-shifts* the time index, hence $x^{(i)}{\Uparrow} = x^{(i+1)}$.
- Up-shifting is extended to expressions: $e{\Uparrow}$ denotes a copy of e where all variables have been up-shifted; in $e{\Uparrow_X}$ only variables X have been up-shifted (hence the suffix $\Uparrow_X$ can be thought of as applying an assignment $\{x^{(i)} \mapsto x^{(i+1)} \mid x \in X, i \in \mathbb{N}\}$).
- We will use $\sigma \in \mathcal{I}(G)^*$ to denote sequences of interactions, and $\pi \in \Lambda^*$ to denote *paths*, i.e., sequences of switch labels. For a given $\pi = \lambda_1 \cdots \lambda_n$, the corresponding sequence of interactions is denoted $\sigma_\pi = \alpha_1 \cdots \alpha_n$.
- A *gate value* for $g \in \mathcal{G}$ is a tuple $g\,\bar{w}$, where $\bar{w} = w_1 \cdots w_n \in \mathcal{U}^*$ is a sequence of values, such that the number of values and their types match with g's interaction variables $i\bar{v}$ (from interaction $(g, i\bar{v}) \in \mathcal{I}(G)$). We write $\mathcal{GU}$ for the set of all gate values.
- Given a gate value sequence $\omega = g_1\,\bar{w}_1, \ldots, g_n\,\bar{w}_n$, we write σ_ω to denote its corresponding interaction sequence such that $(g_i, i\bar{v}_i) \in \mathcal{I}(G)$ for $1 \le i \le n$.
- Because $i\bar{v}$ is fixed for each g (via its interaction $(g, i\bar{v}) \in \mathcal{I}(G)$), a gate value $g\,\bar{w}$ uniquely gives rise to a valuation $\vartheta_{g\,\bar{w}}$ mapping each iv_i to the corresponding w_i.

A path condition only contains interaction variables as free variables. Values of model variables are completely determined by their initial value in the initial location, plus assignments in subsequent switches. We construct a symbolic assignment function a^{ini}_π that maps model variables to expressions over interaction variables (on path π) and symbolic initial values as assigned by *ini*. Values of context variables remain constant along the path, as fixed by the initial assignment *ini*. We use upshifting to distinguish between interaction variables used on path π, since switches may use the same interaction variables.

Definition 4 (Path condition) *Given a BDDTS, for arbitrary paths $\pi \in \Lambda^*$ and initial assignments $ini \in \mathcal{T}(\emptyset)^V$, the symbolic assignment $a^{ini}_\pi \in \mathcal{T}(IV)^V$ and path condition $\eta^{ini}_\pi \in \mathcal{T}(IV)_{\mathsf{Bool}}$ are defined inductively over the length of π:*

$$a^{ini}_\epsilon = ini \qquad \eta^{ini}_\epsilon = \mathbf{true}$$
$$a^{ini}_{\pi\cdot(\alpha,\phi,a)} = a[a^{ini}_\pi{\Uparrow}] \qquad \eta^{ini}_{\pi\cdot(\alpha,\phi,a)} = \eta^{ini}_\pi{\Uparrow} \wedge \phi[a^{ini}_\pi{\Uparrow}]$$

The intuition behind a path condition is that it is fulfilled precisely by those sequences of gate values that satisfy the successive switch conditions. For example, a path condition for the sequence of red switches in Figure 3b is:

$$\begin{aligned}&\mathsf{badge}^1 = ini(\mathsf{P_{badge}}) \wedge ini(\mathsf{A_{badge}}).\mathsf{contains}(\mathsf{badge}^1)\\ &\wedge \mathsf{P_{badge}} = 1234 \wedge ini(\mathsf{A_{badge}}).\mathsf{contains}(ini(\mathsf{P_{badge}}))\\ &\wedge \mathbf{true} \wedge \mathsf{door.id} = 1 \wedge \mathsf{command} = \mathsf{OPEN}\end{aligned}$$

To formulate the symbolic semantics of a BDDTS, we collect goal paths: all paths π to some goal location gl sharing an interaction sequence σ. The goal implication then expresses that for each goal path (π, gl) of interaction sequence σ, the path condition of π should make the output guard of gl true. Intuitively, this expresses that taking all switches of π should

make the output guard at the end true; in BDD terms, this means that the postcondition of the ***Then*** step is satisfied after executing the scenario.

Definition 5 (*EC* **and** *GI*) *Let $\mathcal{B}$ be a saturated BDDTS and $ini \in \mathcal{T}(\emptyset)^V$ an initialization. The* execution condition function $EC_{\mathcal{B},ini}\colon \mathcal{I}(G)^* \to \mathcal{T}_{\mathsf{Bool}}(V)$ *and* goal implication function $GI_{\mathcal{B},ini}\colon \mathcal{I}(G)^* \to \mathcal{T}_{\mathsf{Bool}}(V)$ *are defined, for any $\sigma \in \mathcal{I}(G)^*$, as:*

$$EC_{\mathcal{B},ini} : \sigma \mapsto IG \wedge \bigvee \{\, \eta^{ini}_{\pi} \mid il \xrightarrow{\pi}, \ \sigma_\pi = \sigma \,\}$$
$$GI_{\mathcal{B},ini} : \sigma \mapsto \bigwedge \{\, \eta^{ini}_{\pi} \Rightarrow OG(l)[a^{ini}_{\pi}] \mid il \xrightarrow{\pi} l, \ OG \downarrow l, \ \sigma_\pi = \sigma \,\}$$

Hence, for every σ, the execution condition EC and goal implication GI are predicates over (upshifted) variables IV. The first characterizes when σ is executable, whereas the second expresses the property that, when σ leads to a goal location l, the output guard of l holds whenever σ is executable (see Example 4.1 and 4.2).

Note that we have defined Definition 5 only for saturated BDDTSs. Though the execution conditions and goal implications themselves can be computed for arbitrary models, their intention is to capture the testing power of BDDTSs (Theorem 2 and Corollary 1); but that correspondence only holds for saturated models. Indeed, the saturation of a BDDTS in general preserves neither EC nor GI.

Now we can use the execution condition and goal implication as symbolic BDDTS semantics. We define that on the level of *sets* $\mathbf{B}$ of BDDTSs. As a convenient notation to restrict such a set to those BDDTSs of which the input guard is satisfied by a given initialization $ini \in \mathcal{T}(\emptyset)^V$, we use:

$$\mathbf{B}\restriction_{ini} = \{\mathcal{B} \in \mathbf{B} \mid [\![ini]\!] \models IG_{\mathcal{B}}\} \ .$$

Definition 6 (Testing equivalence) *Two sets of saturated BDDTSs $\mathbf{B}_1, \mathbf{B}_2$ are* testing equivalent, *denoted $\mathbf{B}_1 \simeq \mathbf{B}_2$, if for all $ini \in \mathcal{T}(\emptyset)^V$ and $\sigma \in \mathcal{I}(G)^*$:*

$$\bigvee_{\mathcal{B}\in\mathbf{B}_1\restriction_{ini}} EC_{\mathcal{B},ini}(\sigma) \equiv \bigvee_{\mathcal{B}\in\mathbf{B}_2\restriction_{ini}} EC_{\mathcal{B},ini}(\sigma)$$
$$\bigwedge_{\mathcal{B}\in\mathbf{B}_1\restriction_{ini}} GI_{\mathcal{B},ini}(\sigma) \equiv \bigwedge_{\mathcal{B}\in\mathbf{B}_2\restriction_{ini}} GI_{\mathcal{B},ini}(\sigma) \ .$$

One of the main results of this paper (Corollary 1) is that testing equivalence guarantees that the derived tests always give the same test verdicts on the same behavior. In other words, sets of BDDTSs that are testing equivalent have the same testing power, i.e., the same capability to accept or reject Systems under Test on the basis of their observable behavior.

4. Disjunction Composition

We now come to the main contribution of this paper: the definition of disjunction composition $\bigtriangledown$ for saturated BDDTSs.

Definition 7 *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two saturated BDDTSs. Their disjunction composition is then defined as $\mathcal{B}_1 \bigtriangledown \mathcal{B}_2 = \langle L, V, G, \to, il, IG, OG\rangle$, where*

- $L = (L_1 \cup \{\bot\}) \times (L_2 \cup \{\bot\})$, *with* $L^\circ = (L_1^\circ \cup \{\bot\}) \times (L_2^\circ \cup \{\bot\})$
- $\to$ *is the relation generated by the following rules:*
  1. *Disjunction for shared interactions:*

$$\frac{l_1 \xrightarrow{\alpha,\phi_1,a_1} l_1' \quad l_2 \xrightarrow{\alpha,\phi_2,a_2} l_2' \quad a_1 \Leftrightarrow a_2}{(l_1,l_2) \xrightarrow{\alpha,(\phi_1\wedge\phi_2),a_1\sqcup a_2} (l_1',l_2')} \ 1$$

  2. *Disjunction for non-shared interactions:*

$$\frac{l_1 \xrightarrow{\alpha,\phi_1,a_1} l_1' \quad l_2 \not\xrightarrow{\alpha}}{(l_1,l_2) \xrightarrow{\alpha,\phi_1,a_1} (l_1',\bot)} \ 2\text{-}1$$

$$\frac{l_1 \not\xrightarrow{\alpha} \quad l_2 \xrightarrow{\alpha,\phi_2,a_2} l_2'}{(l_1,l_2) \xrightarrow{\alpha,\phi_2,a_2} (\bot,l_2')} \ 2\text{-}2$$

- $il = (il_1, il_2)$
- $IG = IG_1 \vee IG_2$
- $OG : (l_1,l_2) \mapsto$
$$\begin{cases} OG_1(l_1) & \text{if } OG_1 \downarrow l_1 \text{ and } OG_2 \uparrow l_2 \\ OG_2(l_2) & \text{if } OG_1 \uparrow l_1 \text{ and } OG_2 \downarrow l_2 \\ OG_1(l_1) \wedge OG_2(l_2) & \text{if } OG_1 \downarrow l_1 \text{ and } OG_2 \downarrow l_2 \end{cases}$$

$\mathcal{B}_1 \bigtriangledown B_2$ *is considered to be* ill-defined *if for any reachable $(l_1,l_2) \in L_1 \times L_2$ there are $l_1 \xrightarrow{\alpha,\phi_1,a_1}$ and $l_2 \xrightarrow{\alpha,\phi_2,a_2}$ such that $a_1 \not\Leftrightarrow a_2$.*

In the sequel, we only consider disjunction compositions that are well-defined.

Some explanation is in order. The locations of a composed BDDTS $B_1 \bigtriangledown B_2$ are tuples (l_1,l_2) of locations from $\mathcal{B}_1$ and $\mathcal{B}_2$, where either l_i can be $\bot$ if the behavior has moved outside $\mathcal{B}_i$. The switch relation for a location (l_1,l_2) is constructed from the switches from l_1 (in $\mathcal{B}_1$) and l_2 (in $\mathcal{B}_2$), using two types of rules. Rule 1 applies if both locations have switches for the same interaction α and compatible assignments ($a_1 \Leftrightarrow a_2$): such switches are synchronized while their guards are conjoined. Note that, due to saturation of the $\mathcal{B}_i$, if one such switch exists, then others are guaranteed to exist (for the same interaction) with complementary switch guards, causing the set of switches derived by Rule 1 to be saturated once more. Rules 2-1 and 2-2, on the other hand, deal with interactions for which either l_1 or l_2 has a switch but the other does not (which includes the case where one of the locations equals $\bot$): then the existing switch is copied from the relevant component. As for the output guard of (l_1,l_2), this is in principle the conjunction of the defined output guards of the l_i according to $\mathcal{B}_i$.

If there is a reachable location (l_1,l_2) with switches featuring the same interaction but incompatible assignments ($a_1 \not\Leftrightarrow a_2$), this invalidates the composition. This means that $\bigtriangledown$ is only partially defined. All our results about disjunction composition are implicitly restricted to cases where it is defined.

Proposition 2 and Proposition 3 state expected, but essential, properties of disjunction composition.

Proposition 2 *Disjunction composition yields a saturated BDDTS.*

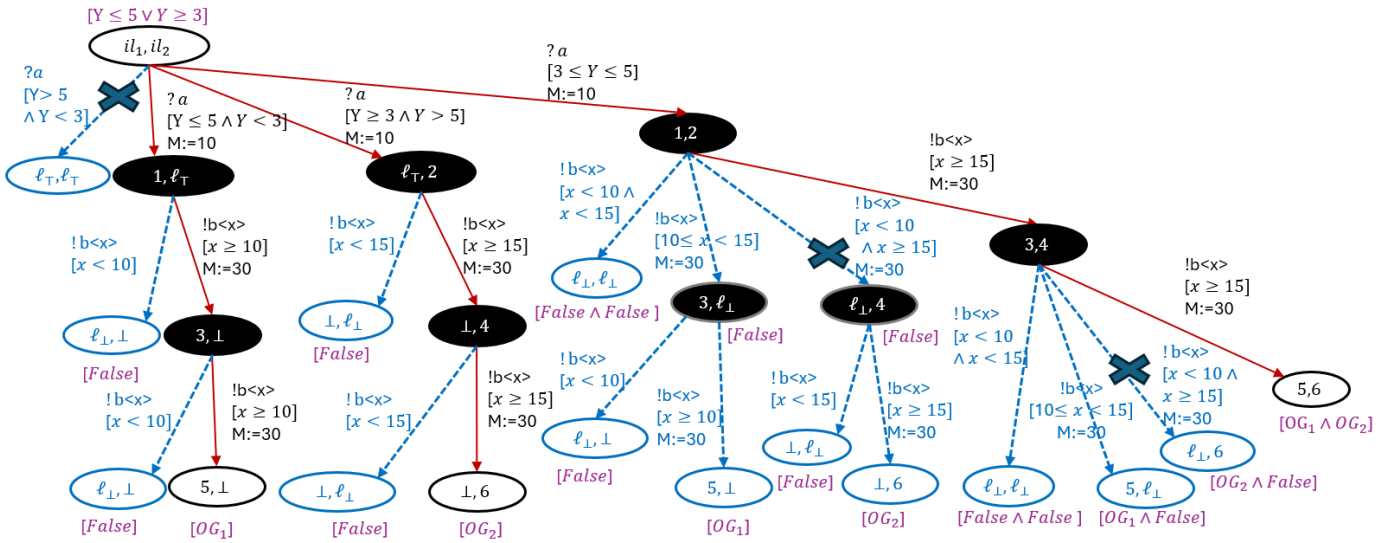


Figure 4 Disjunction composition $B_1 \bigtriangledown B_2$ (see B_1 and B_2 in Figure 2). We mark switches with an unsatisfiable guard with a cross.

Proposition 3 *Disjunction composition is commutative and associative.*

Example 4.1 *Figure 4 shows the disjunction composition of two BDDTS B_1 and B_2. As an example we also build the EC and GI for σ =?a!b!b in B_1, the EC and GI for B_2 can be constructed similarly:*

$$\begin{aligned} EC_{B_1,ini}(\sigma) &= (ini(Y) \le 5 \wedge x^{(1)} \ge 10 \wedge x \ge 10) \\ &\quad \vee (ini(Y) \le 5 \wedge x^{(1)} \ge 10 \wedge x < 10), \\ GI_{B_1,ini}(\sigma) &= ((ini(Y) \le 5 \wedge x^{(1)} \ge 10 \wedge x \ge 10) \Rightarrow OG_1) \\ &\quad \wedge ((ini(Y) \le 5 \wedge x^{(1)} \ge 10 \wedge x < 10) \Rightarrow \mathbf{false}) \end{aligned}$$

We can now state our first main theoretical result: the composition of two BDDTSs is testing equivalent to the two original BDDTSs. This is the justification of the construction: by using the composed $\mathcal{B}_1 \bigtriangledown \mathcal{B}_2$ rather than the individual $\mathcal{B}_i$, we can derive a combined test with exactly the same testing power.

Theorem 1 *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two saturated BDDTSs. Then it holds that:*

$$\{\mathcal{B}_1, \mathcal{B}_2\} \simeq \{\mathcal{B}_1 \bigtriangledown \mathcal{B}_2\} \ .$$

Example 4.2 *Below, we show a small part of EC and GI for composition of Figure 4. Note that for σ =?a!b!b, the EC and GI of individual BDDTSs are equivalent to the composition as stated in Theorem 1.*

$$\begin{aligned} EC_{B_1 \bigtriangledown B_2,ini}(\sigma) &= (ini(Y) < 3 \wedge x^{(1)} \ge 10 \wedge x \ge 10) \\ &\quad \vee (ini(Y) < 3 \wedge x^{(1)} \ge 10 \wedge x < 10) \\ &\quad \vee \ldots \vee (3 \le ini(Y) \le 5 \wedge x^{(1)} \ge 15 \wedge x \ge 15) \\ GI_{B_1 \bigtriangledown B_2,ini}(\sigma) &= ((ini(Y) < 3 \wedge x^{(1)} \ge 10 \wedge x \ge 10) \Rightarrow OG_1) \\ &\quad \wedge ((ini(Y) < 3 \wedge x^{(1)} \ge 10 \wedge x < 10) \Rightarrow \mathbf{false}) \\ &\quad \wedge \ldots \wedge ((3 \le ini(Y) \le 5 \wedge x^{(1)} \ge 15 \wedge x \ge 15) \\ &\qquad \Rightarrow OG_1 \wedge OG_2) \end{aligned}$$

5. Concrete Semantics and Test Cases

In this section we show how to construct test cases from arbitrary BDDTSs (not just saturated ones). As outlined in Figure 1, this is a three-step process: first we translate a BDDTS to a so-called *symbolic transition system* (STS) to encode the input and output guards; then we use the standard semantics for STSs to obtain a *labeled transition system* (LTS); finally, the $\overset{\circ}{\bullet}$-nature of locations is used to convert this into a *test case*.

Definition 8 (STS) *A* Symbolic Transition System *(STS) is a tuple $\mathcal{S} = \langle L, V, G, \rightarrow, il, ini \rangle$, where the first five components are the same as those of a BDDTS, with the proviso that $V = MV$ (hence $CV = \emptyset$), and $ini \in \mathcal{T}(\emptyset)^V$ represents the initial (syntactical) assignment of the location variables.*

For translating BDDTSs into STSs, several approaches were informally discussed in (Zameni et al. 2024), using different ways to obtain the values of the context variables when the output guards are checked. One of those approaches, which adds extra check-and-retrieve switches, was worked out in (Zameni et al. 2025). Here we present a different approach: we *assume* that the domain provides output interactions with interaction variables that retrieve the up-to-date values of the context variables when they are needed. In particular, every switch leading up to a goal location is assumed to have such special interaction variables. This enables output guards to be evaluated as part of the incoming switch guards. In addition, we assume that every transition leading to a goal location originates in a closed location. We call BDDTSs that satisfy this assumption *output-rich*.

Definition 9 (Output-rich) *A BDDTS $\mathcal{B}$ is* output-rich *if*

1. *for all $(g, iv_0 \cdots iv_n) \in \mathcal{I}(G_o)$, there is an injective partial renaming $\rho^g : CV \hookrightarrow \{iv_0, \cdots, iv_n\}$, and*
2. *for all $t \in \rightarrow$, if $OG \downarrow tl_t$ then $sl_t \in L^\bullet$ and $\rho^{g_t} \downarrow x$ for all $x \in CV \cap vars(OG(tl_t))$.*

Though these assumptions make our BDDTS-to-STS conversion less general than the approach of (Zameni et al. 2025), the assumptions are in fact fulfilled in well-known architectures such as APIs with a request-response communication pattern, like our case study (Section 6). In such domains, our approach is more concise.

Definition 10 *Let $\mathcal{B}$ be an output-rich BDDTS with renaming ρ, and let $ini \in \mathcal{T}(\emptyset)^V$ such that $[\![ini]\!] \models IG$. The STS of $\mathcal{B}$ for ini is given by:*

$$STS(\mathcal{B}, ini) = \langle L, V, G, \rightarrow_c, il, ini \rangle \text{ where:}$$
$$\rightarrow_c = \{(sl_t, \alpha_t, \phi_t, a_t, tl_t) \mid t \in \rightarrow, OG \uparrow tl_t\}$$
$$\cup \{(sl_t, \alpha_t, \phi_t \wedge OG[\rho^{g_t}], a_t, tl_t) \mid t \in \rightarrow,\ OG \downarrow tl_t\}$$

Example 5.1 *Figure 5a shows the STS for the saturated BDDTS of Figure 3b. In this example, we asume that ini:* $\mathsf{P_{badge}} := 1234$, $\mathsf{A_{badge}} := [1234, 5678, 0666]$, $\mathsf{AccessGranted} := \mathsf{False}$, $\mathsf{Door}(\mathsf{id} := 1, \mathsf{state} := \mathsf{CLOSED})$ *and* $\rho(\mathsf{Door}) = \mathsf{door}$. *There are two locations with output guards in its BDDTS Figure 3b:* $OG(2) = \mathsf{Door.state} = \mathsf{OPEN}$ *and* $OG(\ell_\perp) = \mathsf{False}$. *In the STS they are both moved and conjoined to the switch guard and in switch with gate* $!\mathsf{trigger}_{\mathsf{door}}$ *the context variable* Door *is substituted with interaction variable* door.

Note that in the test case of Figure 5b, we deliberately do not change the value of context variable Door *at location* 2 *to keep it consistent with the intuition of context variables in the BDDTS (their value is not changed by the model); instead, their initial value is accessible through the system under test. The actual value of the Door at that point in the execution is carried by the interaction variable* door*, which is supplied by the domain on retrieval and checked against the output guard as part of the incoming switch guard. The check is therefore performed against the fresh value, even though the context-variable slot itself is never rewritten.*

Note that in our setting, there is no need to reassign former context variables once they become model variables in the STS, since their values are already obtained through interaction variables and the corresponding model variables are not used afterwards. The next step is to translate ($^\circ_\bullet$-natured) STSs into ($^\circ_\bullet$-natured) LTSs, as in (Van den Bos & Tretmans 2019). First we define the general notion of LTS.

Definition 11 (LTS) *A* labeled transition system *(LTS) is a tuple $\mathcal{L} = \langle Q, A, \rightarrow, q_0 \rangle$, where:*

- *Q is a set of states with designated initial state $q_0 \in Q$*
- *A is a set of labels, partitioned into input labels A_i and output labels A_o*
- *$\rightarrow \subseteq Q \times A \times Q$ is a transition relation.*

Next, we recall the STS semantics in terms of LTSs from (Van den Bos & Tretmans 2019), adding the $^\circ_\bullet$-nature.

Definition 12 *Let $\mathcal{S} = \langle L, V, G, \rightarrow, il, ini \rangle$ be an $^\circ_\bullet$-natured STS. The* semantics *of $\mathcal{S}$ is defined as the $^\circ_\bullet$-natured LTS $[\![\mathcal{S}]\!] = \langle L \times \mathcal{U}^V, \mathcal{GU}, \rightarrow', (il, [\![ini]\!]) \rangle$ with $\mathsf{N}(l, \vartheta) = \mathsf{N}(l)$ for all $(l, \vartheta) \in L \times \mathcal{U}^V$, and*

$$\rightarrow' = \{((sl_t, \vartheta), u, (tl_t, [\![a_t \sqcup id_t]\!]_{\vartheta_u \sqcup \vartheta})) \mid q \in L \times \mathcal{U}^V, u \in \mathcal{GU}, t \in \rightarrow, \vartheta_u \sqcup \vartheta \models \phi_t\}$$

where $id_t(v) = v$ for all $\{v \in V \mid a_t \uparrow v\}$ and $id_t \uparrow v$ otherwise.

In the final step, $^\circ_\bullet$-natured LTSs are converted into *test cases*, defined as LTSs with designated *pass* and *fail* sink states.

Definition 13 *A* test case *is tuple $\langle Q, A, \rightarrow, q_0, P, F \rangle$ where $\langle Q, A, \rightarrow, q_0 \rangle$ is an LTS and $P, F \subseteq Q_{Sink}$ are special sets of* pass *and* fail *states.*

A test case TC is evaluated as follows: the behavior of a SuT is said to *pass* TC if it leads *through* a state in P and to *fail* it if it leads through a state in F. If a system neither passes nor fails (never reaching a state in $P \cup F$), the test is said to be *inconclusive*. Formally:

$$TC \text{ passes } \omega \ \Leftrightarrow\ \exists \xi \preceq \omega : q_0 \xrightarrow{\xi} P$$
$$TC \text{ fails } \omega \ \Leftrightarrow\ \exists \xi \preceq \omega : q_0 \xrightarrow{\xi} F\ .$$

To construct a test case from an $^\circ_\bullet$-natured LTS $\langle Q, A, \rightarrow, q_0 \rangle$, we introduce a fresh sink state $q_\mathbf{f}$, which is the only *fail* state, to which we add new transitions from each closed state $q \in Q^\bullet$ for every action $a \in A_o$ that is not enabled from q. All open sink states in Q are *pass*. As an example, Figure 5b shows the test case of BDDTS $\mathcal{B}$ from Figure 3b. Note that because BDDTSs are deterministic, so are the resulting STSs, LTSs and test cases.

Definition 14 *Let $\mathcal{L} = \langle Q, A, \rightarrow, q_0 \rangle$ be an $^\circ_\bullet$-natured LTS and $q_\mathbf{f} \notin Q$. The test case for $\mathcal{L}$ is defined by $TC(\mathcal{L}) = \langle Q \cup \{q_\mathbf{f}\}, A, \rightarrow', q_0, Q^\circ_{Sink'}, \{q_\mathbf{f}\} \rangle$ with*

$$\rightarrow' = \rightarrow \cup \{(q, a, q_\mathbf{f}) \mid q \in Q^\bullet, a \in A_o, q \not\xrightarrow{a}\}$$

This chain of transformations from BDDTSs to test cases then yields:

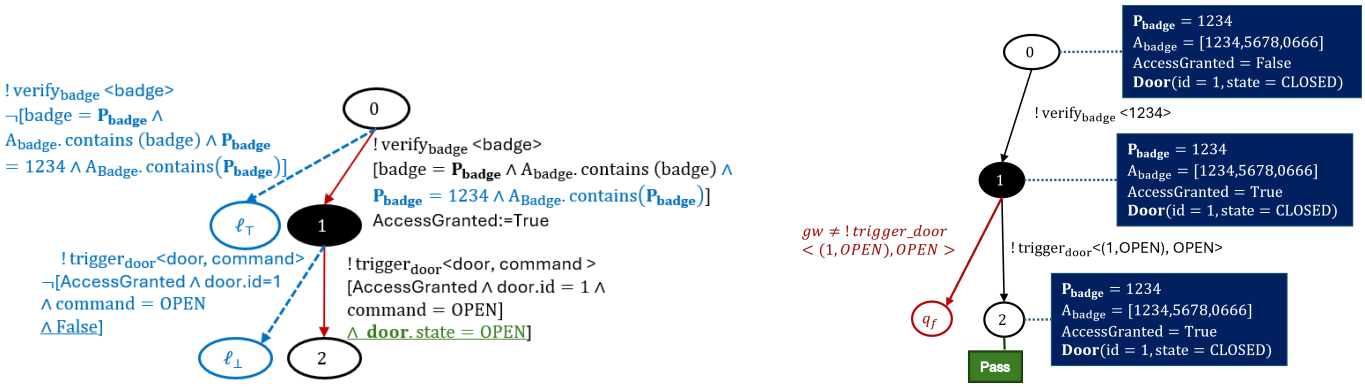
Definition 15 *The test case for an output-rich BDDTS $\mathcal{B}$ with initialization $ini \in \mathcal{T}(\emptyset)^V$ such that $[\![ini]\!] \models IG$ is defined as:*

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

Note that, in contrast to the symbolic semantics and the disjunction composition, the test case construction does not require BDDTSs to be saturated. It is therefore important to note that saturating as in Definition 3 does not change the fail verdicts of the derived test case.

Proposition 4 *If $\mathcal{B}$ is an output-rich BDDTS and $ini \in \mathcal{T}(\emptyset)^V$ such that $[\![ini]\!] \models IG$, then for any gate value sequence $\omega \in \mathcal{GU}^*$*

$$\omega \text{ fails } TC(\mathcal{B}, ini) \ \Leftrightarrow\ \omega \text{ fails } TC(\mathcal{B}^{sat}, ini)\ .$$



(a) STS for BDDTS $\mathcal{B}$ of Figure 3b

(b) Test case for STS of Figure 5a

Figure 5 Translation from BDDTS to STS to test case

Unfortunately, Proposition 4 does not extend to pass verdicts: it turns out that saturation may in fact change the pass verdicts of the derived tests, either by turning a pass verdict into inconclusive or the other way around. This is a shortcoming that we plan to address in the future.

Our second main result, Theorem 2, relates the symbolic semantics of BDDTSs (Definition 5) to its concrete semantics in terms of test cases (Definition 14).

To state this formally, we need some additional notation. For a valuation $\vartheta \in \mathcal{U}_\perp^{IV}$, with $\Uparrow\vartheta \in \mathcal{U}_\perp^{IV\Uparrow}$ we denote a variant that applies ϑ to an upshifted domain. This is formally defined by:

$$\Uparrow\vartheta : x \mapsto \vartheta(y) \quad \text{if } x = y\Uparrow \ .$$

(Note the difference with $a\Uparrow$, in which the upshift is applied to the *result* of a.) A gate value sequence ω then gives rise to a valuation ϑ_ω for interaction variables that are increasingly upshifted for "older" interactions, inductively defined by

$$\vartheta_\epsilon = \emptyset \qquad \vartheta_{\omega\, u} = \Uparrow\vartheta_\omega \sqcup \vartheta_u \ .$$

To also evaluate goal implications, we further extend ϑ_ω to context variables by using the correspondence defined by ρ (g is the output gate of last switch):

$$\hat{\vartheta}_\omega = \vartheta_\omega \sqcup (\vartheta_\omega \circ \rho^g) \ .$$

Theorem 2 characterizes when a gate value sequence ω passes or fails a test case derived from a BDDTS $\mathcal{B}$, in terms of $\mathcal{B}$'s execution conditions and goal implications (Definition 5). Intuitively, a gate value sequence passes the test case if and only if the execution conditions and goal implications hold for all prefixes of the sequence, up to the point where the test case can no longer proceed—because no continuation satisfies the execution conditions. In this sense, ω represents a "complete" run of the test. On the other hand, ω fails the test case if there is a prefix of ω so that its execution conditions hold, i.e. the path in $\mathcal{B}$ can be taken, but the goal implication at the end is not satisfied.

Theorem 2 (Correctness of Symbolic Semantics) *Let $\mathcal{B}$ be a saturated output-rich BDDTS and let $ini \in \mathcal{T}(\emptyset)^V$ such that $[\![ini]\!] \models IG_\mathcal{B}$. For any gate value sequence $\omega \in \mathcal{GU}^*$, the following holds:*

1. *ω* passes *$TC(\mathcal{B}, ini)$ if and only if there is a $\xi \preceq \omega$ such that $\vartheta_\xi \models EC_{\mathcal{B},ini}(\sigma_\xi)$ and*
   (a) *for all $\bar{\xi} \preceq \xi$, $\hat{\vartheta}_{\bar{\xi}} \models GI_{\mathcal{B},ini}(\sigma_{\bar{\xi}})$ and*
   (b) *for all $u \in \mathcal{GU}$, $\vartheta_{\xi u} \not\models EC_{\mathcal{B},ini}(\sigma_{\xi u})$.*
2. *ω* fails *$TC(\mathcal{B}, ini)$ if and only if there is a $\xi \preceq \omega$ such that $\vartheta_\xi \models EC_{\mathcal{B},ini}(\sigma_\xi)$ and $\hat{\vartheta}_\xi \not\models GI_{\mathcal{B},ini}(\sigma_\xi)$.*

The following encapsulates the last main theoretical contribution of this paper. It follows directly from Theorem 2 and the definition of $\simeq$ (Definition 6).

Corollary 1 (Testing equivalence preserves test verdicts) *Let $\mathbf{B}_1, \mathbf{B}_2$ be sets of saturated output-rich BDDTSs, and let $ini \in \mathcal{T}(\emptyset)^V$. If $\mathbf{B}_1 \simeq \mathbf{B}_2$, then for all value sequences $\omega \in \mathcal{GU}^*$:*

$$\begin{aligned} &\exists \mathcal{B} \in \mathbf{B}_1\restriction_{ini} : TC(\mathcal{B}, ini) \text{ fails } \omega \\ &\qquad \Leftrightarrow \ \exists \mathcal{B} \in \mathbf{B}_2\restriction_{ini} : TC(\mathcal{B}, ini) \text{ fails } \omega \\ &\forall \mathcal{B} \in \mathbf{B}_1\restriction_{ini} : TC(\mathcal{B}, ini) \text{ passes } \omega \\ &\qquad \Leftrightarrow \ \forall \mathcal{B} \in \mathbf{B}_2\restriction_{ini} : TC(\mathcal{B}, ini) \text{ passes } \omega \ . \end{aligned}$$

6. Case study: train information boards

We applied our theory of disjunction composition to an industrial example from NS, the Dutch Railways (Nederlandse Spoorwegen), with the aim of demonstrating that the disjunction composition process can be automated. We note that this case study was of an exploratory nature and hence not meant as a comprehensive evaluation.

Our case study is a subsystem from NS, responsible for the information displayed on boards regarding train arrivals and departures. NS uses BDD scenarios to verify that the system can accurately retrieve and present the stored information from a database on the boards. Various combinations of train arrivals and departures can occur at a platform, making it essential to test these combinations thoroughly. To test this level of complexity, NS wrote some rather lengthy scenarios, in the

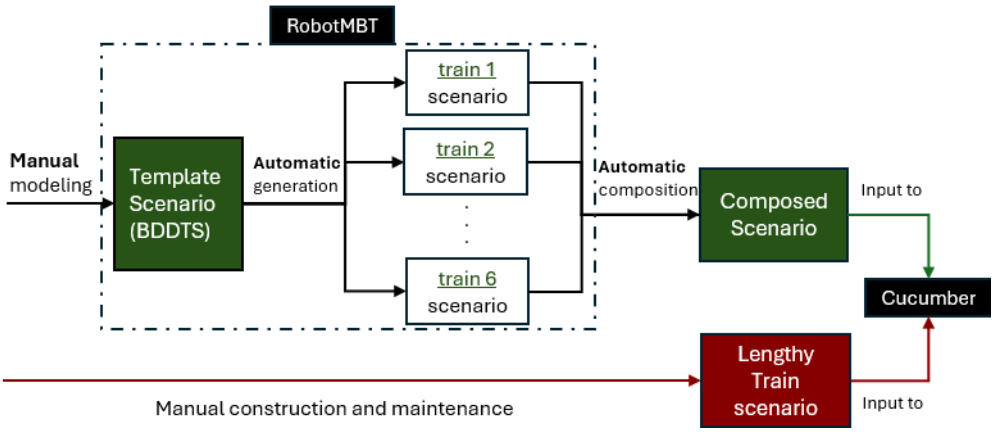
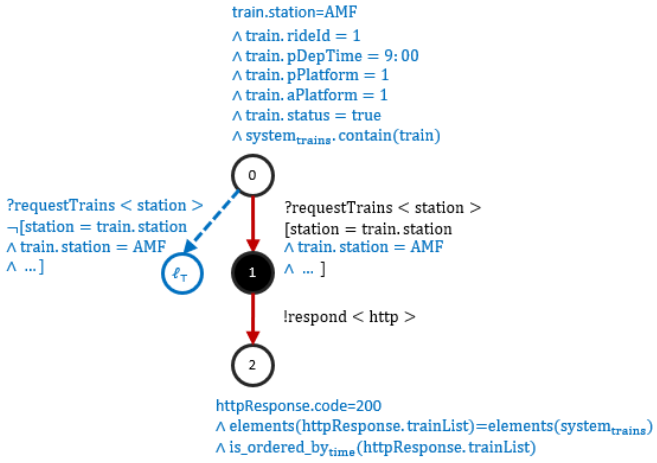


Figure 6 Compositional MBT+BDD approach vs. the NS BDD approach

Scenario 4 Requesting departing trains
- ***Given*** a departing train at station "AMF" with rideId 1
  And the train has a planned departure time of 08:00
  And the train has a planned departure on platform 1
  And the train has current departure platform 1
- ***Given*** . . . (continues with X other trains)
- ***When*** I request the departing trains with station "AMF" and platform "1"
- ***Then*** I get a 200
  And I get X+1 trains back
  And the 1st departing train contains departure time "08:00"
  And...
- ***When***...(requesting all platforms)
- ***Then***...(the number of trains back and ordered by time)

(a) Format of manual NS BDD scenarios for several trains



(b) BDDTS for one train

Figure 7 Format of manual BDD scenario and constructed BDDTS

format of Figure 7a. In particular, the ***Given*** step is quite lengthy, because specific trains need to be each spelled out. With the current manual scenarios, always trains with the same station, departure time, etc. are used.

The manual NS scenarios suffer from three issues. First, their length and complexity make them hard to read and maintain. Second, data diversity is low: the same concrete values (station "AMF", departure time "8:00", etc.) are reused across runs. Third, the combinations of trains at a platform are picked by hand, so they tend to reflect what the engineer already expected to happen. We tackle the first two issues with a standard model-based step. We make a *template BDDTS*, a symbolic single-train abstraction similar to one represented in (Figure 7b), but with the difference that instead of having fixed values, the model defines the allowed values, e.g. train ids in range $\{1, \ldots, m\}$, n stations, k platforms, etc ($m, n, k \in \mathbb{N}$). Concrete scenarios are then generated from this template, so instead of maintaining many near-duplicate scenarios with concrete values, the engineer only maintains the abstract template model.

So far, this is within reach of parameterized testing. The composition operator takes us a step further toward addressing the third issue. We apply *disjunction composition* on the copies of the template, which gives us a BDDTS that symbolically captures all admissible multi-train situations at a station, not only the ones someone thought to write down. Generating from this composed model produces combinations of ids, departure times, and platforms that are not anticipated and can potentially reveal unexpected behavior.

Figure 6 shows both workflows side by side. In the NS approach (red, below), a single lengthy scenario is constructed and maintained manually and handed to Cucumber. In our approach (green, above), only the template BDDTS is built manually; from it, RobotMBT automatically generates one scenario per train and then automatically composes them into a single scenario, which is then handed to Cucumber. The template itself (Figure 7b) captures the intended behaviour symbolically, namely that the response must contain exactly the stored trains, ordered by departure time, rather than tying this to a concrete value such as "8:00" as in Figure 7a.

Since no available tool directly implements disjunction composition, we combined an existing library, RobotMBT, and a custom developed script. To be able to use the RobotMBT library as-is, we first used RobotMBT, to generate BDD scenarios with concrete data values from our template BDDTS; secondly, we applied disjunction composition on those BDD scenarios with the script. Each of the two is explained in more detail below.

For generating test cases with varied data values, we used the open-source Robot Framework library RobotframeworkMBT (introduced in (Artifact 2025)). RobotMBT extends the open-source Robot Framework (Robot Framework 2026) with model-based testing capabilities. Robot Framework uses keyword-driven testing and supports BDD-style Given–When–Then scenarios, where each step is implemented as a keyword. We expressed our BDDTS of Figure 7b in RobotMBT, by using the modeling functionality of RobotMBT. Then we defined ranges for symbolic values (e.g., range(1,6) for train identifiers, and similar for departure times or platforms) using the tool's modifier functionality. RobotMBT then automatically generates multiple concrete scenarios from these ranges. Hence, these scenarios represent the copies of the template BDDTS with chosen data values.

We partially implemented disjunction composition, for the BDDTS with data values, by developing a script for extracting only the 'conjunction trace', i.e., the right most branch of the disjunction composition that conjoins the guards of all ?requestTrains switches. This way we obtain the most interesting test case of the disjunction composition, namely the one that combines all trains in a single scenario, which is comparable to the original NS scenario of Figure 7a. We furthermore did some minor conversions, so that the resulting test case had a suitable format for Cucumber, so that it could be executed in that tool, just like the original NS scenario.

We generated test cases with 6 trains, i.e. some of the trains of the NS scenario. Though our data generation was random, we note that it could also generate all the values occurring in the original NS scenario. We executed the generated Cucumber tests with different trains until we found a bug.

The bug showed up as follows: the last train was expected to have a departure time of 9:25, but the system returned 9:10. Careful debugging showed that the correct time had in fact been saved; it was the testing environment around the system that supplied the wrong time. The bug had gone unnoticed with the existing manual scenarios, which always stored trains with the same fixed values and in the same fixed groupings.

From these exploratory experiments, we draw the following conclusions. We showed that a *BDD testing process* can be improved with *model-based testing* and *disjunction composition* through our approach, by building a prototype implementation on top of existing tooling. With this setup, we generated test cases that combine the request-response commands for any number of trains, rather than the fixed number in the original NS scenario, and with varying data values rather than the fixed ones. This led to the discovery of a bug that the existing BDD-based testing at NS had not caught. We also note that the template BDDTS is considerably smaller than the original NS scenario, which makes maintenance and updates easier as requirements evolve.

7. Related Work

Below we discuss related work on formal modeling and composition of BDD scenarios, and on composition operators for other transition systems.

Modeling of BDD scenarios. (De Biase et al. 2024) generate SysML state machines from gherkin requirements, and (Wiecher et al. 2020) provide a language in Kotlin for writing BDD scenarios for automotive system validation. Though clearly having the same aim in mind, a key difference is that these works do not employ formal composition.

Composition of BDD scenarios. (Kannengiesser et al. 2020) propose representing BDD scenarios for Cyber-Physical Production Systems as a specific kind of state machine and composing those that react to the same event. (Kang & Silva 2023) translate BDD scenarios into Timed Automata, allowing for manual composition where scenarios share ***When*** steps. Our disjunction composition also composes shared ***When*** steps. However, neither of these works addresses conflicting ***Then*** steps, like we do.

Other composition operators for BDD scenarios, orthogonal to our paper, are discussed in the following works. In (Silva 2023), a domain-specific language (DSL) is used to abstract from BDD steps. The DSL allows finer-grained scenarios to be embedded within each step, effectively enabling scenario linkage at different abstraction levels. Formal sequential composition of BDD scenarios is introduced in (Zameni et al. 2025). This composition enables subsequent testing of several BDD scenarios. The SkyFire tool automatically generates BDD scenarios from UML diagrams (Li et al. 2016). The resulting scenarios tend to be long and are connected in sequence, reflecting the structure of the source UML models. Test cases are then derived using the Cucumber framework.

Composition operators for transition systems. Various composition operators have been studied for Labeled Transition Systems (LTSs). In (Beneš et al. 2015), conjunction, parallel composition, and quotient operators are discussed, with the parallel composition later refined in (Daca et al. 2014). Also, (Janssen 2020) presents a conjunction operator for LTS, capturing intersection over outputs. We apply the same idea defined differently via saturation and disjunction, and on BDDTS which is a symbolic model.

Parallel composition has been studied in the context of ioco-theory in (Van der Bijl et al. 2004). In (Van Cuyck et al. 2023) and (Van Cuyck et al. 2024) a notion of mutual acceptance is introduced, such that components can be tested instead of their parallel composition. While in our work the same actions may synchronize, as in parallel composition, we do not consider interleaving; this is an orthogonal concept. Additionally, our BDDTSs describe behaviors, i.e. functionality, which may be unrelated to the (component) structure of the tested system.

The transformation of STS into LTS is not a core contribution of this paper; instead, we rely on the concept of path conditions from (Van den Bos & Tretmans 2019). It should be noted that there are alternatives, such as (de Boer & Bonsangue 2021) that could have been used for this purpose.

Finally, we should acknowledge the huge body of work on composition of LTSs — essentially, the field of Process Algebra (Bergstra et al. 2001). Though this is clearly the source of inspiration for our work, it is in lifting the concepts to the symbolic level that the contribution of this paper lies.

8. Conclusion and Future Work

In this work, we presented the disjunction composition operator for BDD Transition Systems (BDDTS)—formal models of BDD scenarios. This operator enables the combination of multiple BDDTSs so that tests can be derived from this integrated model, instead of only testing the scenarios in isolation. We have defined a symbolic semantics for BDDTS, to show correspondence between two BDDTS and their composition. We also show the relation between this symbolic semantics and the concrete semantics in terms of test cases.

In the scope of this paper, we have ignored certain aspects of testing such as the analysis of the (root) cause of a failure. Though a failed test can clearly be traced back to the BDDTS path, this is not enough when the BDDTS itself is obtained through (disjunction) composition: one would also want to trace it back to the original BDD scenario that was not fulfilled. However, we expect no particular difficulties here, as every composed BDDTS switch is derived from the component switches through the derivation rules in Definition 7. It is straightforward to build this dependency into the tooling.

One aspect not discussed in this paper is the complexity of the composition. While the number of states grows linearly in the number of components, the initial fan-out, i.e., the number of transitions from the initial state, may grow exponentially: every initial transition is the combination of transitions from all components. Since the cases we explored in practice involve only a few components, we do not expect this to cause problems. If the model grows beyond reasonable bounds, one can apply standard MBT techniques, where tests are generated while the specification is computed as far as needed. Since full model coverage is typically infeasible anyway, such an exploration goes hand in hand with the choices made in test generation.

For future work, we aim to address the fact that saturation can confuse pass and inconclusive verdicts, and investigate techniques to preserve them. We also plan to develop more composition operators for BDDTS, e.g. for loops and parallel composition. Furthermore, we would like to generalize our symbolic semantics to more generic models than BDDTS. Finally, we aim to do more elaborate case studies, and also further integrate the disjunction composition in MBT tooling.

Acknowledgments

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.

This research was supported by NWO project OCENW.M.23.155 *Evidence-Driven Black-Box Checking (EVI)*.

References

Beneš, N., Daca, P., Henzinger, T. A., Křetínský, J., & Ničković, D. (2015). Complete composition operators for IOCO-testing theory. In *Proceedings of the 18th international ACM SIGSOFT Symposium on Component-Based Software Engineering* (p. 101–110). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2737166.2737175

Bergstra, J. A., Ponse, A., & Smolka, S. A. (Eds.). (2001). *Handbook of process algebra*. Elsevier.

Chelimsky, D., Astels, D., Helmkamp, B., North, D., Dennis, Z., & Hellesoy, A. (2010). *The RSpec book: Behaviour Driven Development with RSpec, Cucumber, and Friends* (1st ed.). Pragmatic Bookshelf. Retrieved from https://dl.acm.org/doi/10.5555/1965448

Cucumber. (2026). Retrieved from https://cucumber.io/docs/guides/

Daca, P., Henzinger, T. A., Krenn, W., & Nickovic, D. (2014). Compositional specifications for IOCO testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation* (p. 373-382). doi: 10.1109/ICST.2014.50

De Biase, M. S., Bernardi, S., Marrone, S., Merseguer, J., & Palladino, A. (2024, Dec 01). Completion of SysML state machines from Given–When–Then requirements. *Software and Systems Modeling*, *23*(6), 1455-1491. doi: 10.1007/s10270-024-01228-3

de Boer, F. S., & Bonsangue, M. M. (2021). Symbolic execution formally explained. *Formal Aspects Comput.*, *33*(4-5), 617–636. doi: 10.1007/S00165-020-00527-Y

Frantzen, L., Tretmans, J., & Willemse, T. A. C. (2005). Test generation based on symbolic specifications. In J. Grabowski & B. Nielsen (Eds.), *Formal approaches to Software Testing* (Vol. 3395, pp. 1–15). Springer. doi: 10.1007/978-3-540-31848-4_1

Janssen, R. (2020). Combining Partial Specifications using Alternating Interface Automata. In H. Wehrheim & J. Cabot (Eds.), *Fundamental Approaches to Software Engineering* (Vol. 12076, pp. 462–481). Springer. doi: 10.1007/978-3-030-45234-6_23

Kang, E.-Y., & Silva, T. R. (2023). Towards Formal Verification of Behaviour-Driven Development Scenarios using Timed Automata. In *2023 30th Asia-Pacific Software Engineering Conference (apsec)* (p. 612-616). doi: 10.1109/APSEC60848.2023.00081

Kannengiesser, U., Krenn, F., & Stary, C. (2020). A Behaviour-Driven Development Approach for Cyber-Physical Production Systems. In *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)* (Vol. 1, p. 179-184). doi: 10.1109/ICPS48405.2020.9274755

Li, N., Escalona, A., & Kamal, T. (2016, April). Skyfire: Model-Based Testing with Cucumber. In *2016 IEEE International Conference on Software Testing, Verification and Validation (icst)* (p. 393-400). IEEE. doi: 10.1109/ICST.2016.41

Reqnroll. (2026). Retrieved from https://docs.reqnroll.net/latest/

Robot Framework. (2026). Retrieved from https://robotframework.org/

Rose, S., & Nagy, G. (2021). *The BDD books — Formulation: Document examples with Given/When/Then*. Leanpub. Retrieved from https://leanpub.com/bddbooks-formulation

Silva, T. R. (2023). Towards a Domain-Specific Language for Behaviour-Driven Development. In *2023 IEEE Sympo-*

sium on Visual Languages and Human-Centric Computing (VL/HCC) (p. 283-286). doi: 10.1109/VL-HCC57772.2023.00054

Tretmans, J. (2008). Model-Based Testing with Labelled Transition Systems. In R. M. Hierons, J. P. Bowen, & M. Harman (Eds.), *Formal Methods and Testing: an Outcome of the FORTEST Network, Revised Selected Papers* (Vol. 4949, pp. 1–38). Springer. doi: 10.1007/978-3-540-78917-8_1

Van Cuyck, G., Van Arragon, L., & Tretmans, J. (2023). Compositionality in Model-Based Testing. In S. Bonfanti, A. Gargantini, & P. Salvaneschi (Eds.), *Testing Software and Systems* (Vol. 14131, pp. 202–218). Springer. doi: 10.1007/978-3-031-43240-8_13

Van Cuyck, G., Van Arragon, L., & Tretmans, J. (2024). Testing Compositionality. In D. Marmsoler & M. Sun (Eds.), *Formal Aspects of Component Software* (Vol. 15189, pp. 39–56). Springer. doi: 10.1007/978-3-031-71261-6_3

Van den Bos, P., & Tretmans, J. (2019). Coverage-Based Testing with Symbolic Transition Systems. In D. Beyer & C. Keller (Eds.), *Tests and Proofs* (Vol. 11823, pp. 64–82). Springer. doi: 10.1007/978-3-030-31157-5_5

Van der Bijl, M., Rensink, A., & Tretmans, J. (2004). Compositional Testing with IOCO. In A. Petrenko & A. Ulrich (Eds.), *Formal Approaches to Software Testing* (Vol. 2931, pp. 86–100). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-540-24617-6_7

Wiecher, C., Japs, S., Kaiser, L., Greenyer, J., Dumitrescu, R., & Wolff, C. (2020). Scenarios in the loop: integrated requirements analysis and automotive system validation. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM. doi: 10.1145/3417990.3421264

Zameni, T. (2025). *Executable BDD scenario models for robotframeworkMBT*. Zenodo. (Software artefact) doi: 10.5281/zenodo.14892704

Zameni, T., Van den Bos, P., Foederer, J., & Rensink, A. (2025). Sequential Composition of BDD Transition Systems for Model-Based Testing. In C. Ferreira & C. A. Mezzina (Eds.), *Formal Techniques for Distributed Objects, Components, and Systems* (Vol. 15732, pp. 36–54). Springer. doi: 10.1007/978-3-031-95497-9_3

Zameni, T., van den Bos, P., & Rensink, A. (2026). *Disjunction composition of BDD transition systems for model-based testing*. (Extended version) doi: 10.48550/arXiv.2602.17237

Zameni, T., Van den Bos, P., Rensink, A., & Tretmans, J. (2024). An Intermediate language to Integrate Behavior-Driven Development Scenarios and Model-Based Testing. In *IEEE international Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE. doi: 10.1109/SANER-C62648.2024.00033

Zameni, T., Van den Bos, P., Tretmans, J., Foederer, J., & Rensink, A. (2023). From BDD Scenarios to Test Case Generation. In *IEEE International Conference on Software Testing, Verification and Validation, ICST* (pp. 36–44). IEEE. doi: 10.1109/ICSTW58534.2023.00019

About the authors

Tannaz Zameni is a PhD candidate at the University of Twente, in the Netherlands. Her research lies in formal model-based testing. She developed a formal semantics for Behavior-Driven Development, aimed at lowering the barrier to applying model-based testing in industry. You can contact the author at tannaz.zameni@gmail.com.

Petra van den Bos is an Assistant Professor at the University of Twente (The Netherlands). Her current research focuses on software quality in general, and on model-based testing specifically. You can contact the author at p.vandenbos@utwente.nl or visit https://petravdbos.nl.

Arend Rensink is a full professor at the University of Twente, in the area of Software Modeling, Transformation and Verification. His focus is on the theory and practice of Graph Transformation; for this purpose, he has created (and maintains) the tool GROOVE. You can contact the author at arend.rensink@utwente.nl or visit https://people.utwente.nl/arend.rensink.