From BDD Scenarios to Test Case Generation

Tannaz Zameni*, Petra van den Bos*, Jan Tretmans†‡, Johan Foederer§, and Arend Rensink*
*Formal Methods and Tools, University of Twente, Enschede, The Netherlands
†Department of Software Science, Radboud University, Nijmegen, The Netherlands
‡ TNO-ESI, Eindhoven, The Netherlands
§ TOPIC Embedded Systems, Best, The Netherlands
Email: *{t.zameni, p.vandenbos, arend.rensink}@utwente.nl, †tretmans@cs.ru.nl, §johan.foederer@topic.nl

***Abstract*—Model-based testing (MBT) offers the possibility of automatic generation and execution of tests. However, it is not yet widely used in industry due to the difficulty in creating and maintaining models. On the other hand, Behavior Driven Development (BDD) is becoming more popular in the agile development process to achieve a common understanding of the system under development among stakeholders and to automate testing. However, BDD scenarios are written in human language and are usually not precise enough. Moreover, tests extracted from BDD scenarios are too short and incomplete; they only cover a very small part of the system. Our goal is to combine these two approaches to benefit from the usability of BDD and the test automation capabilities of MBT. In this paper, we first define a formal model of scenarios that we call BDD Transition Systems, second, we create more complete tests by composing scenarios (model composition), and finally, we generate and execute tests automatically. We demonstrate the applicability of this approach in a real-world example: an industrial printer.**

***Index Terms*—Behavior-Driven Development, Model-Based testing, Compositional testing**

I. Introduction

Modern software systems are ever-growing in size and complexity, offering an ever wider range of functionalities, and increasingly connecting to their environment. Systematic testing plays a major role in getting confidence in the quality of such systems. Software testing, however, is often an error-prone, expensive, and time-consuming process. Estimates are that testing consumes 30-50% of the total software development costs. The tendency is that the effort spent on testing is still increasing due to the continuing quest for better software quality, and the ever-growing size, complexity, and connectivity of systems. The situation is aggravated by the fact that the complexity of testing tends to grow faster than the complexity of the systems being tested, in the worst case even exponentially. This may seriously hamper the testing of future generations of software systems, implying that smarter, more effective, and more efficient testing methods are required.

a) MBT: Model-Based Testing (MBT) is one of the technologies that are propagated to meet these testing challenges. MBT is a form of black-box testing where the model serves as a specification for the system under test (SUT), prescribing the behaviour that the SUT shall, and shall not exhibit. The main advantage of MBT is that it enables the automated, algorithmic generation of large amounts of valid test cases including corresponding expected results. MBT, in particular, MBT using formal models, originates from research on formal methods and testing. Nowadays, a reasonable number of commercial and open-source tools for MBT are available, but, despite its solid foundations and promises of automated test generation, there is no widespread use of MBT in the industry, yet.

The main bottleneck that prohibits the broad application of MBT is the construction and availability of the appropriate behavioural models for MBT. Firstly, there is some reluctance against investment in making models, as companies see this as having to develop and maintain yet another software artifact. Secondly, mastering the art of behavioural modeling requires abstract thinking, education, and experience that is not always available. Thirdly, the information necessary to construct a model, in particular for legacy, third-party, or outsourced systems or components is not always (easily) available. And last but not least, the specialized languages in which MBT models are expressed do not excel in readability and understandability for non-experts, such as product owners, customers, and other stakeholders. This complicates communication with these stakeholders, and it does not facilitate obtaining feedback and validating MBT models, i.e., getting confidence that the model really models what was intended.

b) BDD: Behaviour Driven Development (BDD) is an agile approach to software development. A key goal of BDD is to foster communication and shared understanding of what the software under development should do, among all stakeholders of the product such as developers, product owners, product analysts, testers, customers, and business developers [1] [2], [3].

In the BDD approach, three activities are distinguished: discovery, formulation, and automation. During *discovery*, the required behaviour of the software or feature under development is explored in structured conversations by all stakeholders involved, by constructing examples of the required behaviourr. Such explorations are sometimes referred to as 'three amigos' sessions. In the *formulation* phase, the examples are documented in structured natural language, in such a way that they are understandable and shared by all stakeholders, which facilitates validation. This is also called *specification by example*. The most popular style to write these documented examples, called *scenarios*, is the *Given*/*When*/*Then* style used in *Gherkin* language. There are other styles like *Context*/*it* in RSpec [4] and tables [5]. The documented examples are written in such a way, that during *automation* phase they can

be transformed into executable test cases. These can then be used to verify whether the developed software indeed satisfies the requirements documented in the scenarios. The collection of scenarios is also referred to as *living documentation*. Automation of Gherkin scenarios is supported by many tools, e.g., Cucumber [6] and SpecFlow [7].

Unlike MBT, BDD does not originate from research but from software engineering practice. Nowadays, many software companies use some form of BDD approach to explore, specify, and automate tests for software features.

c) BDD and MBT: Among the strong points of the BDD approach are the collaborative exploration of the requirements by making examples, the documentation of examples in scenarios expressed in structured, readable natural language, and the readability and understandability of scenario specifications by all stakeholders. The lack of such a shared understanding of readable specifications is a weak point of current MBT approaches.

On the other hand, MBT provides a solid foundation in the form of formal semantics and well-defined testing theory, leading to algorithmic test generation of many, long, diversified, and valid tests, together with test result analysis. In addition, the formal underlying theory enables reasoning about concurrency, non-determinism, model coverage, and compositionality. Most of these aspects are weak points of BDD. There is no underlying theory providing formal semantics to scenarios, the size and number of scenario-based test cases are limited, i.e., they usually, and deliberately, test one particular aspect, and not a combination of aspects or features. Concurrency, non-determinism, and model coverage are not considered. Composing scenarios is sometimes used, but in a very informal, ad-hoc, and sometimes ambiguous way, for example, two scenarios with overlapping *Given*-conditions sometimes mean that a choice can be made, and sometimes that both should be considered concurrently or conjunctively. Also, the infamous Gherkin *And*-keyword can have different meanings: sometimes it means sequence, sometimes concurrency, and sometimes logical *and* (conjunction). Additionally, reaching a state that satisfies a particular *Given*-condition can be difficult. This is currently completely left to the implementer of the test code in a so-called *step*-definition. Then it might be, however, that laying some other scenarios head-to-tail will easily reach such a state. There is no way to reason about such compositions of scenarios in the BDD approach. In MBT theory, this corresponds to the standard problem of reachability analysis.

Given this analysis of the strong and weak points of BDD and MBT, the goal of our research is to combine them in such a way that we obtain their complementary strengths. Basically, this means that we combine the exploration and specification construction in the form of Gherkin scenarios from BDD, with the test generation, compositionality, and formal reasoning from MBT. We aim to accomplish this by transforming Gherkin scenarios into small models in a formal MBT modeling language. In this way, we can use the discovery and formulation phases of BDD to construct scenarios that are readable and understandable. Moreover, after transforming these scenarios into these small models in the MBT formalism, we can compose these small models into larger models, we can use reachability analysis to reach particular *Given*-states, and we can generate many, long, diversified, and valid test cases that test different aspects and combined features. To the best of our knowledge, there is no research found in the literature that automatically generates, composes, and executes tests from BDD scenarios based on a formal model with formal semantics.

In this paper, we define BDD Transition Systems (BDDTS) as Symbolic Transition Systems (STS) with preconditions (for *Given* steps) and postconditions (for *Then* steps). These BDDTS are our formal MBT modeling language. STS is a well-defined formalism for MBT [8], that supports formal reasoning, composition, and test generation [9], [10]. We show how scenarios in *Given*/*When*/*Then*-style are transformed into simple BDDTSs using a real-world example of an industrial printer. Then we elaborate on how these simple BDDTSs can be composed into larger BDDTSs, which are the basis for test generation following [10].

We concentrate on the sequential composition of BDD scenarios and show this by example, that is the post-condition of one scenario (*Then*-step) enables the pre-condition of the next scenario (*Given*-step), where 'enables' means logical implication. Many other forms of composing scenarios, or, actually, composing BDDTSs generated from scenarios, are possible, e.g., the choice between scenarios (disjunction), concurrent scenarios, conjunctive scenarios, and sequential composition if there is no implication but just overlap, or interrupting scenarios. These compositions and the full formal definition of sequential composition will be considered in future papers.

d) Overview: The next section introduces BDD scenarios in general, the running example of the industrial printer, and the printer's scenarios. Section III defines the formalism of BDDTS, after which Section IV illustrates the translation from BDD scenario to BDDTS using the real-world industrial printer example. Section V discusses the composition of BDDTSs and test generation from the composed BDDTS. Section VI and VII present related work, conclusions, and future research.

II. BDD Scenarios

This section defines the structure of the BDD scenarios and provides a real-world industrial example of a set of scenarios.

A. Structure of BDD scenarios

The structure of BDD scenarios given below is adapted from [3]. In Section III, we will give a definition of a formal model for BDDs corresponding to the below definition.

A Behavior-Driven Development scenario describes a small function of the system. It describes some user-visible behavior of the system. This user can be a human user, another component in the same system, or another system interacting with the system. We refer to all these users as the *system*

environment or simply *environment*. We will call the system being described with the scenarios the *System Under Test* (because we will generate test cases), or just the *system*. A BDD scenario consists of three parts: *Given*, *When*, and *Then*.

Given specifies the required system state that is a precondition for the next step (*When*).

When describes an action or a sequence of actions. Either the environment or the system performs such an action. The action has an effect or consequence on the party not performing the action.

Then can be described in three different ways

- it describes the action the system does after the *When* step, or
- it describes the state in which the system ends up finally, or
- it describes both, i.e., the action the system performs after *When*, *and* the final state.

We note that if *Then* only specifies the action, there is still an implicit final state, namely the state reached after performing the action.

B. Printer example

We now introduce the running example of this paper: a printer. The printer works as follows. The operator starts by **submitting** a *job file* using a *submission method*. Based on the submission method, the printer **adds** a *controller job* to the print queue called *scheduled jobs*. At the time the controller job is added to the scheduled jobs the printer **starts printing**. If the operator does nothing, the printer continues and **completes printing**. But, while the controller job is being printed the operator may **pause** printing. A paused job may be **resumed**, i.e. the printer continues printing the job from the scheduled jobs, or **moved** to another queue called the *waiting jobs*. Waiting jobs are not printed. The operator can **move** the controller job from the waiting jobs to the *scheduled jobs* to start printing the job from scratch.

To describe this flow with BDD scenarios, the flow is divided into several scenarios that describe a small functionality. Below, we write out all these BDD scenarios.

C. Scenarios

Scenario 1: A controller job is added to the scheduled jobs after a job is submitted

- Given a Job file
- When the operator submits the Job file with ⟨Submission method⟩
- Then the printer adds a new Controller job to the scheduled jobs
- And the Controller job is of type ⟨job type⟩

The table below shows the values allowed for *Submission method* in combination with *job type*:

Submission method	Job Type
LPR	Production job
IPP	Production job
JMF	Production job
Socket	Streaming job

Scenario 2: A controller job is moved to the printed jobs the moment printing completes

- Given a controller job is in the scheduled jobs
- When the printer starts printing the controller job
- And the printer completes printing the controller job
- Then the controller job is in the printed jobs

Scenario 3: There is a hard copy of the controller job after completing the printing of the controller job.

- Given a controller job is in the scheduled jobs
- When the printer starts printing the controller job
- And the printer completes printing the controller job
- Then there is a printed output
- And the printed output is a hard copy of the controller job

Scenario 4: While a job is being printed, it can be paused

- Given a controller job is printing
- When the operator pauses the printing of the controller job
- Then the controller job is paused

Scenario 5: A job that is paused can be resumed to be printed

- Given a controller job is paused
- When the operator resumes printing the controller job
- then the controller job is printing

Scenario 6: A controller job that is paused and moved to the waiting jobs before it completes, is not moved to the printed jobs

- Given a controller job is paused
- When the operator moves the controller job to the waiting jobs before the printer completes printing
- Then the controller job is in the waiting jobs
- And the controller job is not in the printed jobs

Scenario 7: A controller job that is in the waiting jobs can be moved to the scheduled jobs

- Given a controller job in the waiting jobs
- When the operator moves the controller job to the scheduled jobs
- Then the controller job is in the scheduled jobs

III. A FORMAL MODEL FOR BDD SCENARIOS

In this section, we define a formal model, namely a transition system, for BDD scenarios. This transition system will need to store the data elements of a scenario, e.g. the controller job of the printer example. These data elements are defined in the next section, and after that, we define the BDD transition system itself.

A. Data elements

In this section, we introduce standard programming concepts like variables, terms (i.e. expressions), types, and assignments. For a complete, formal definition of data elements, we refer to [10].

a) Syntax: Terms consist of ground terms, e.g., $true$, variables, e.g., 'x', and operations, e.g., '$\wedge$'. Let X be a set of variables. The set of terms consisting of variables X is denoted as $\mathcal{T}(X)$. Ground terms are the terms without variables, denoted as $\mathcal{T}(\emptyset)$.

Terms have a type, e.g. term $true \wedge x$ has type $Bool$. With $true \wedge x : \mathcal{T}_{Bool}(\{x\})$ we denote that term $true \wedge x$ is of type $Bool$ and contains (at most) variable x. We assume that there is a function $type$ mapping any term to its type, e.g. $type(true \wedge x) = Bool$. For any set of variables X, we define $\mathcal{T}(X)$ to only contain well-formed and well-typed terms, i.e., the non-well-formed term $true\wedge$ and the non-well-typed term $4 \wedge 3$ are not in $\mathcal{T}(X)$.

An assignment assigns a term to a variable, e.g. in $x := x + 1$, term $x + 1$ is assigned to variable x. Given some sets X and Y of variables, $\mathcal{T}(Y)^X$ denotes any set of assignments, where each term $t \in \mathcal{T}(Y)$ is assigned to a variable $x \in X$.

b) Semantics: A *valuation* $\vartheta(X)$ is a function assigning values to variables X. Ground terms have a value corresponding to their syntax, e.g. the value of $true$ is denoted as `true`. A *term evaluation* $\vartheta_{eval}(X)$ extends valuation $\vartheta(X)$ to evaluate terms containing variables. For example, if $\vartheta(x) = \texttt{false}$, then $\vartheta_{eval}(\{x\})(true \wedge x) = \texttt{false}$.

Given a term evaluation $\vartheta_{eval}(Y)$, a set of assignments $A \in \mathcal{T}(Y)^X$ is evaluated to a valuation $\vartheta(X)$. Here, $\vartheta(X)$ is defined as the evaluation of each assignment $x := t \in A$, such that $\vartheta(X)(x) = \vartheta_{eval}(Y)(t)$. For example, given $\vartheta(\{y\})_{eval}(y+1) = 3$, assignment set $\{x := y+1\}$ evaluates to $\vartheta(\{x\})$ with $\vartheta(\{x\})(x) = 3$.

B. BDD transition systems

We provide the definition of a Symbolic Transition System inspired by [10].

Definition 1. *A Symbolic Transition System is a tuple* $S = \langle\ LOC, l_0, \mathcal{V}, i, \mathcal{I}, \Lambda, \rightarrow\ \rangle$*, where*

- LOC is a set of locations.
- $l_0 \in LOC$ is the initial location.
- $\mathcal{V}$ is a set of global variables. They are global and accessible in the entire transition system.
- $i \in \mathcal{T}(\emptyset)^{\mathcal{V}}$ is the initial assignment of the global variables.
- $\mathcal{I}$ is a set of interaction variables. We assume $\mathcal{V} \cap \mathcal{I} = \emptyset$ and set $Var =_{def} \mathcal{V} \cup \mathcal{I}$. They are called interaction variables as they represent the data interaction associated with a switch (see below). Variables *(Var)* have types, i.e. either basic data types like *Bool, Int* and *String* or composite datatypes with different fields of different types.
- Λ is the set of gates. *We define* $\Lambda = \Lambda_i \cup \Lambda_o$ where Λ_i, Λ_o are the sets of input and output gates, respectively.
- $\rightarrow \subseteq LOC \times \Lambda \times \mathcal{I}^* \times \mathcal{T}_{Bool}(Var) \times \mathcal{T}(Var)^{\mathcal{V}} \times LOC$ is the switch relation.
  In a switch $(loc, \lambda, f_0...f_k, \varphi, \rho, loc') \in \rightarrow$ the elements are called (source) location, gate, interaction variables, guard, assignments, and (destination) location, respectively.

A BDD transition system (BDDTS) extends the definition of STS with a few additional elements: it is a tuple $B = \langle S, L_g, IG, OG, guardOfLoc\ \rangle$, where

- S is the Symbolic Transition System
- $L_g \subseteq LOC$ is the set of goal locations.
- $IG : \mathcal{T}_{Bool}(\mathcal{V})$ is the input guard of the BDDTS, denoting the pre-condition of the initial location.
- $OG \subseteq \mathcal{T}_{Bool}(\mathcal{V})$ is the set of output guards denoting the post-conditions of the BDDTS.
- $guardOfLoc : L_g \rightarrow OG$ is a function mapping goal locations to their corresponding output guard.

C. Semantics of BDDTS

The semantics of an STS is formally defined in [10]. With respect to semantics, BDDTS differs from STS by the added input guard and output guards on locations. In this subsection, we provide a short explanation of the intuition of the semantics of STS and BDDTS.

Initially, the global variables of an STS have the values as determined by initialization i, and the current location in l_0. We note that the values of any global variable can be obtained in any location. Next, we can execute an *enabled* switch. Let $(loc, \lambda, f_0...f_k, \varphi, \rho, loc')$ be a switch of the STS. This switch is enabled if the current location is loc, and the evaluation of guard φ is `true` for current values of the global variables and the values of the interaction variables. The values of the interaction variables are determined by the environment (if λ is an input gate) or by the system (if λ is an output gate). Execution of an enabled switch results in evaluating the assignments ρ of the switch. This way, global variables may be assigned new values. Additionally, the current location becomes loc'. As an example consider the STS of Figure 2. Initially, we are in location 0 and suppose that for Job File (JF), $JF.id = 0$ and the input guard $is(JF)$ evaluates to `true`. Suppose that we wish to execute the switch with input gate $?submit$. By letting the environment choose $id = 0$, we have that guard $id == JF.id$ evaluates to `true`, so we may indeed execute this switch. Then the assignment for Submission Method (SM), $SM := sm$ is executed accordingly. The next switch from location 1 to 2 can then be executed similarly, though the system now chooses the values of the interaction variables. For example, if the system uses controller job cj with $cj.type == "Streamingjob"$, the environment should have chosen $sm := "Socket"$ in the previous step, to enable the execution of the switch to location 2. If so, the assignments are executed, we reach location 2, and we can check whether the output guard really holds (as will be explained in Subsection V-B).

IV. BDD SCENARIO TRANSLATION

In this section, we define how to convert a BDD scenario written in Given-When-Then style to a BDD transition system. Currently, we do this conversion manually. We restrict ourselves to what we explained to be a BDD scenario in section 3. We identify the elements of the BDD transition system from

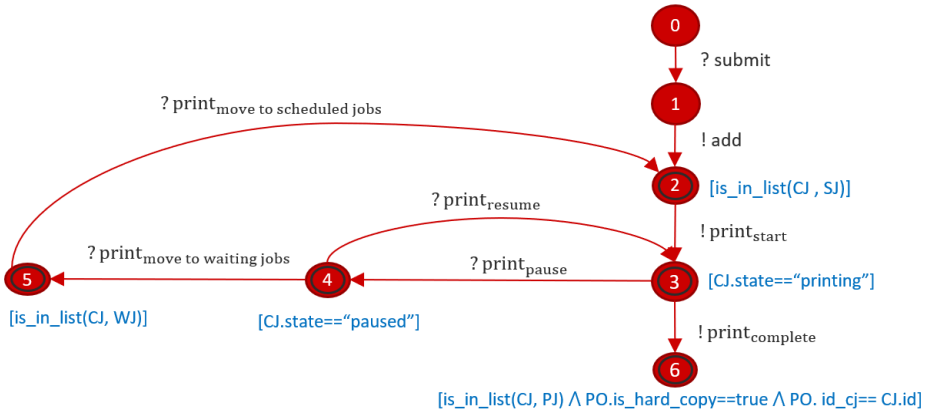

Figure 1. Partial Specification of printing a job file in a printer only showing switch gates and output guards of composed BDDTS

the scenarios. We use Scenario 2 of Subsection II-C to explain the translation.

From the ***Given*** step of Scenario 2 we extract the pre-conditions for the scenario. This precondition describes the required state of the system. We need two elements: the global variables($\mathcal{V}$) and the input guard(IG).

- Given a *controller job* is in the *scheduled jobs*

In this scenario, the controller job (CJ) and the scheduled jobs (SJ) are the global variables. We write global variables in capital letters. The fact that the controller job is in the scheduled jobs is the condition to be checked, so we define operation is_in_list, and define $IG = is\_in\_list(CJ, SJ)$ as the boolean term defining the input guard of the initial location.

From the ***When*** step we extract the actions performed by the system or the environment. Each action is translated to a gate. If *When* describes multiple actions, conjuncted by *and*, we build a sequence of switches for these actions. If the actor is the environment, we use an input gate, and if the actor is the system we use an output gate. In addition, we look for interaction variables and global variables. The interaction variables update the value of global variables with assignments.

- When the printer **starts printing** the *controller job*
- And the printer **completes printing** the *controller job*

Here, $\text{print}_{\text{start}}$ and $\text{print}_{\text{complete}}$ are both output gates and the controller job is the global variable CJ. The $\text{print}_{\text{start}}$ gate is on the switch from the initial location 0 to location 1 and the $\text{print}_{\text{complete}}$ gate on a switch from location 1 to location 2. The guards are defined on interaction and global variables to ensure the conditions on data are satisfied and the value of global variables are updated in the assignment. There are two interaction variables *id,state* for gate $!print_{start}$, the *id* is used in the switch guard to make sure the printer is printing the requested controller job and *state* is used to check if the state of the job is changed to "printing" after $\text{print}_{\text{start}}$. The interaction variable *state* is then assigned to the state field of the global variable CJ.

In the ***Then*** step, we look for the global variables and output guards:

- Then the *controller job* is in the *printed jobs*.

controller job and printed jobs are global variables CJ, PJ, and is_in_list(CJ, PJ) is the output guard.

We note that the information we extract from a single scenario might be insufficient for the model. We obtain complementary information from the related scenarios in the set of existing scenarios. Scenarios 2 and 3 are examples of this case. There are two main outputs from the system when the printing is completed: 1) The controller job appears in the printed jobs and 2) there is a hard copy of the controller job. These are defined in separate scenarios but they are both needed to have a sufficient set of variables for the action $!print_{complete}$. To treat ambiguities in scenarios, we add extra information to the model. For example, the relation between Printed output (PO) and the corresponding Controller job (CJ) in scenarios 2 and 3 is defined by storing the id of CJ in PO, PO.id_cj==CJ.id.

In Figures 2-8 you find the BDDTS for scenarios 1-7. The BDDTS of Scenario 1 is defined as $B_1 = \langle\langle LOC, l_0, \mathcal{V}, , i, \mathcal{I}, \Lambda, \rightarrow \rangle, L_g, IG, OG, guardOfLoc \rangle$ where:

$LOC = \{0, 1, 2\}$ $\quad\quad$ Λ_i ={?submit}
$l_0 = 0$ $\quad\quad$ Λ_o={!add}
$\mathcal{V}$ = {JF, SM, CJ,SJ } $\quad\quad$ L_g={2}
$\mathcal{I}$ = { jf, sm, cj, sj} $\quad\quad$ IG = is(JF)

$\rightarrow$= { (0, ?submit, ⟨ jf, sm⟩, id==JF.id, SM:=sm, 1),
(1, !add,⟨ cj, sj⟩, ϕ_1[1] ,{CJ:=cj, SJ:=append(CJ,SJ)},2) }

i= {JF:Job(id:=0), SM:=null, CJ:=null, SJ:=Queue(name="Scheduled Jobs", elements= EmptyList))}
OG = is_in_list (CJ , SJ) ∧ (CJ.type=="Production Job" ∨ CJ.type =="Streaming job")

V. Composition and Testing

A. Pre-post condition composition

In this section, we explain by example how to compose BDDTS and then extract test cases from this composition.

[1] ϕ_1 is the guard on the switch from location 1 to 2, shown in Figure 2

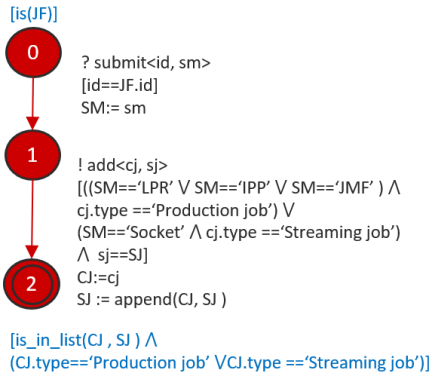

Figure 2. BDDTS for scenario 1

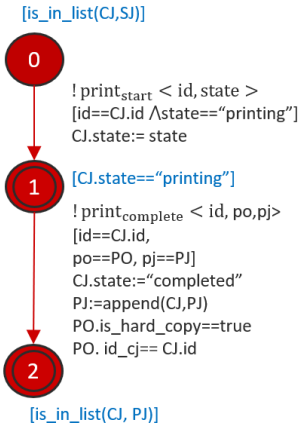

Figure 3. BDDTS for scenario 2

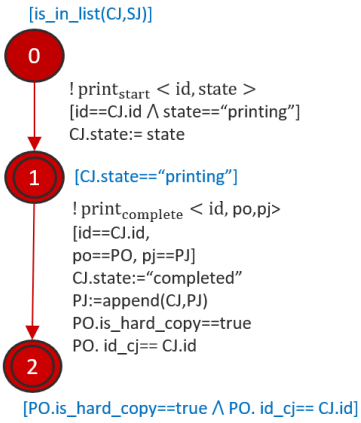

Figure 4. BDDTS for scenario 3

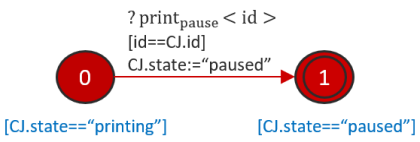

Figure 5. BDDTS for scenario 4

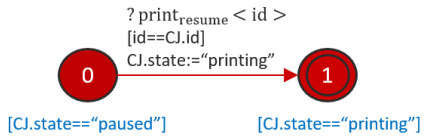

Figure 6. BDDTS for scenario 5

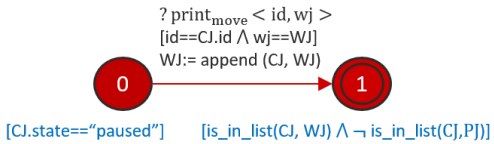

Figure 7. BDDTS for scenario 6

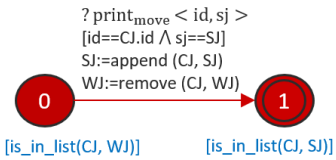

Figure 8. BDDTS for scenario 7

We use a pre-post-condition composition, such that the input guard of scenario B is satisfied by an output guard of scenario A. The composition composes BDDTS A and B sequentially, by merging the respective goal location of A with the initial location of B. Specifically, we define *composability* of two BDDTS as follows:

Let A and B be two BDDTS, where l is a goal location of A, and IG is the input guard of B. Then B is composable with A in l if the output guard of l in A implies the input guard of B, i.e. for all valuations $\vartheta(\mathcal{V}_A \cup \mathcal{V}_B)$, we have that $\vartheta(\mathcal{V}_A \cup \mathcal{V}_B)_{eval}(guardOfLoc(l) \implies IG) = \texttt{true}$.

Take scenarios 1 and 2 in Figures 2, and 3 as examples. The input guard of scenario 2 is *is_in_list(CJ, SJ)*, and the output guard of scenario 1 is $is\_in\_list(CJ, SJ) \wedge (CJ.type == "ProductionJob" \vee CJ.type == "StreamingJob")$. The implication holds since the input guard of scenario 2 is the left term of the conjunction of the output guard of scenario 1. Another example is that the input guard of scenario 4 is the same as the output guard of scenario 5 in Figures 5and 6.

We now explain the pre-post-composition by example and defer giving a general definition to future work. We compose the BDDTS of all scenarios of Subsection II-C. The end result is shown in Figure 1. We start with scenario 1, as we assume the input guard of this scenario to be true in the initial location of a printer. Scenario 1 is composable with both scenarios 2 and 3, because the output guard of scenario 1 is stronger than both input guards. We pick scenario 2, and merge the goal location of scenario 1 with the initial location of scenario 2, i.e. the print$_{start}$ switch can now be taken from the goal location of scenario 1. We choose to set the output guard of the goal location to be the weaker input guard of scenario 2 (see later in example why).

We note that the BDDTSs of scenarios 2 and 3 are the same, except for the output guard of the last location. We can therefore conjunct the output guard of scenario 3 to goal location 2 of scenario 2, which is now location 6 in the composition.

Location 3 of the composition is the same as the input guard of scenario 4, so trivially, scenario 4 can be composed now. We merge scenario 4 with location 3 of the composition. Since the output guard of location 3 is the same as the input guard of scenario 4, we just omit the input guard.

Similarly, the output guard of location 4 is the same as the input guard of scenario 5, so we add it to the composition. We note that the output guard of scenario 5 is the same as the output guard of location 3 of the composition, so we merge the respective locations such that the switch with the $\text{print}_{\text{resume}}$ gate loops back to location 3.

Similarly, scenarios 6 and 7 are added. The output guards of scenario 7 and location 2 are the same, so we make the switch of scenario 7 reach location 2. Note that this would not have been possible if we didn't weaken the output guard of location 2 to only be is_in_list(CJ,SJ), since the output guard of scenario 7 would then be weaker than the output guard of location 2, such that merging would violate composability.

We have the following remarks on composing BDDTS:

- The precondition of a BDD scenario can be inconsistent with the rest of the scenario (e.g. by mistake). Translation to BDDTS then preserves this inconsistency. For example, if the input guard of scenario 2, that the controller job is in the scheduled jobs, would be omitted, this could imply that the state of this job cannot be 'printing', such that the guard of the switch with the $\text{print}_{\text{start}}$ gate is violated. Hence, this should be checked before composition, as composition relies on the validity of input guards.
- Similarly, a mistake may be present in the output guard, e.g. the guard could be unsatisfiable. For example, if location 1 of scenario 4 would have output guard CJ.state == "printing" this is inconsistent with the assignment CJ.state="paused" of the previous switch. Composing a BDDTS in a goal location with an unsatisfiable output guard is pointless, and should therefore be avoided by checking the satisfiability of output guards beforehand.
- Output guards could be strengthened by including the restrictions that are imposed by previous guards and assignments of switches (i.e. *path condition* in [10]), allowing more BDDTS to be composed in the respective goal location.
- There are edge cases where composition with weakening the output guard and merging locations may lead to inconsistencies, comparable with the discussed inconsistencies for input guards. However, weakening is a preferred property for pre-post-composition, as it allows more scenarios to be composable.

B. Testing

To generate test cases from BDDTS, we use the test generation algorithm as described in [10]. This algorithm generates test cases that reach all switches of an STS. Specifically, this means that all scenarios will be executed, and all output guards will be checked as part of a test case. To use the algorithm, we need to translate a BDDTS to an STS. Specifically, we need to encode the input guard and output guards in an STS. We note that, according to the above composition, we may assume that the initial location of a BDDTS corresponds to the initial location of the system being tested. Therefore we do not need to check the input guard as the variables are initially assigned and the input guard holds. Hence, we only need to check the output guards in test cases.

We call the STS extracted from the BDDTS the *test model*. In this test model, we substitute the output guards in every goal location of the BDDTS by two special switches with gates ?check and !retrieve. We assume that we can obtain values of global variables from the system. By the ?check gate the tester requests the values of the variables used in the output guard, from the system. This is done by providing a value, via an interaction variable, that identifies the variable we ask for, e.g. the id of a controller job. The system can then respond through the interaction variables of the !retrieve switch, by providing the actual value of the requested variables, e.g. a controller job with fields id=0, type="Production Job", and state="printing". The switches of the check? and receive! gates are encoded as a loop from and to the goal location.

Figure 9 shows the test model of scenario 1. Compared to its BDDTS in Figure 2, the ?check and !retrieve switches and the intermediate loop location are added with dashed lines. In the ?check switch we use the interaction variables to pass identifiers for retrieving the controller job and scheduled jobs list from the system. In the !retrieve switch we then check with the guard that the returned values have the same identifiers and adhere to the condition specified by the output guard.

In Subsection V-A we noted that the user may write a post-condition for a BDD scenario that cannot be satisfied for any value of global variables. For example, if the output guard of location 4 (i.e. scenario 4) would have been CJ.state == 'printing', this would be inconsistent with the assignment CJ.state="paused" of the previous switch $\text{print}_{\text{pause}}$. As a consequence, the test generation algorithm of [10] will then not be able to generate a test that reaches the receive! switch of this output guard. Hence, this way we are able to notice this inconsistency and notify the user of their mistake.

With the algorithm of [10] we could obtain a test case for scenario 1 that reaches the !retrieve gate. The gates and values for a successful execution would look as follows:
?submit ⟨ Job(id=0), JMF ⟩
!add ⟨ CJob(id=0, type='Production Job', state='ready'), Queue(name='Scheduled Jobs', elements= EmptyList) ⟩
?check ⟨0, 'Scheduled Jobs'⟩
!retrieve ⟨ CJob(id=0, type='Production Job', state= 'paused') , Queue(name='Scheduled Jobs', elements=List(id=0) ⟩

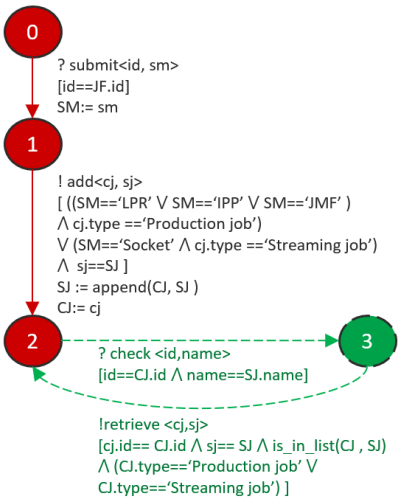

Figure 9. Test model for scenario 1

VI. Related Work

We divide related work into four categories: model-based approaches for scenarios, model-based testing with STSs, testing based on BDD, and model-based testing with BDD.

a) Model-based approaches for scenarios: There is quite a history of (semi) automatic model generation from scenarios like [11] [12] [13] In these papers, scenarios are expressed as sequence diagrams (SD) and then converted to state charts. Due to the lack of precise formal semantics for UML diagrams, in [11], they use extra tooling like OCL, for defining variables and Finite State Machines (FSM), such that this extra information enables converting SDs to state charts and merge scenarios. In [12] and [13] FSMs are used as well. In [14] [15] they focus on testing, and transform UML models into Labelled Transition systems, to add precise formal semantics, similar to those above.

In comparison, we use BDD scenarios, written in a structured text format. The text format has made it popular among non-technical stakeholders *in industry*. BDD scenarios focus on the behavior of the SUT, with steps that specify preconditions, actions, and expected behavior. These scenarios describe how the system should behave in response to various inputs and conditions while SDs only specify the sequences of actions. However, just like UML, BDD scenarios might be ambiguous. We address this by formalizing them with BDDTS. STSs are better suited for modeling complex systems. LTSs and FSMs lack the notion of data. Although in our approach scenario translation is currently manual, automation is possible by e.g. using parsers as in [16].

b) Model-based testing with STSs: In [9], Frantzen et al. introduce Symbolic Transition Systems, that extend Labelled Transition Systems with data. They provide a test algorithm based on the $ioco_{\mathcal{F}}$ relation. [10] is an extension of Frantzen's work that provides robust test selection based on switch coverage. In this paper, we built on the STS definition and test generation algorithm of [10].

c) Testing based on BDD: "The difficulty of writing system-level test cases" is one of the challenges presented in [17]. Our approach helps in this regard by model composition and automatic generation of tests from BDD scenarios. In [18] they combine testing and formal verification by integrating test scenarios and formal properties in a single human-readable document. Then they use the Cucumber [6] tool for testing using the document. In contrast, we convert scenarios into STSs, which are per se formal models, and generate tests from the model. In [19] they provide a technique for regression testing in BDD. Their technique finds and selects the test code that is likely to be affected and needs to be modified for a change in the system. By composing scenarios leading to the code change, we could achieve the same goal and have high traceability between the tests and scenarios. In [16], the authors introduce a semi-automatic approach for extracting the code skeleton and step definition from a single scenario. They create class and sequence diagrams in a semi-automatic way and they have implemented this in the Cucumber tool. While we currently do the translation from scenarios to BDDTS manually, we focus on model (scenario) composition for a more comprehensive set of test cases.

d) MBT with BDD: In [20], an MBT tool called *Skyfire* is presented. *Skyfire* automatically generates Cucumber test scenarios from UML state machine diagrams. The tests are then generated by the Cucumber tool. A similar approach is taken in [21]. They use UML diagrams to generate acceptance tests in the form of sequences of Gherkin scenarios. Executable test cases are then generated from these scenarios. This is different from what we do. We convert scenarios to formal models and generate tests from the model rather than the scenarios. In [22] they provide technical integration of BDD with the MBT tool *Graphwalker* and the *Robot Framework*, but they provide no formalism. In [23], a combination of acceptance test-driven development and model-based testing is presented in some real-world projects. They conclude that both approaches complement each other and increase test coverage. In our work, we provide an intertwined approach to benefit from both BDD scenarios and formal models. In [24] and [25], the authors use BDD to automate the assessment of artifacts throughout the development process. They use computational ontologies to formalize the concepts used in scenarios and generate test cases from ontology models, while we generate tests from the formal BDDTS.

VII. Conclusions and Future Work

In this paper, we proposed an approach for automatic test generation and execution of BDD scenarios. We introduce a formal model for BDD scenarios: BDD transition system (BDDTS). For a set of real-world BDD scenarios of an industrial printer, we show how to translate BDD scenarios into BDDTS, and how to compose these BDDTS, with respect to the pre-and postconditions of scenarios. To automatically generate test cases, we convert the composed BDDTS to a Symbolic Transition System by adding special switches with ?check and !retrieve gates, for checking postconditions of

BDD scenarios with the System Under Test. We use the test generation algorithm from [10] to obtain test cases, such that all scenarios are executed, and all postconditions are checked. There are several directions for future work:

- Find a general, formal definition of pre-post condition composition. This composition should allow the precondition of a scenario to be weaker than the postcondition of the scenario to be composed while preventing the introduction of inconsistencies.
- Support a check and possibly provide a correction for inconsistencies caused by the writer of a BDD. Also, BDD postconditions can be strengthened by taking into account the consequences of previous actions of the scenario itself and other scenarios in the (partial) composition. This way, more scenarios can be composed.
- In this paper, the composition is performed before test generation and execution. However, the composition could also be dynamic: scenarios are then composited 'on the fly' during test generation and execution. The advantages are that test execution can be steered based on past execution results and the tester's current wishes.
- We provided our definition of BDD scenarios in Subsection II-A as we found most BDD scenarios are written vaguely and not suitable for translation to BDDTS. The next step is automatic correction and modeling of BDD scenarios. This could be implemented in existing tools like Cucumber and SpecFlow.
- Finally, a combination of pre-post-composition with other forms of composition, like parallel composition, conjunction, and disjunction, should be investigated.

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 (partly) financed by the Dutch Research Council (NWO).

REFERENCES

[1] D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy, *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends*, 1st ed. Pragmatic Bookshelf, 2010. [Online]. Available: https://dl.acm.org/doi/10.5555/1965448

[2] G. Nagy and S. Rose, *The BDD Books - Discovery Explore behaviour using examples*. Leanpub, 2018. [Online]. Available: https://leanpub.com/bddbooks-discovery

[3] ——, *The BDD Books – Formulation Document examples with Given/When/Then*. Leanpub, 2021. [Online]. Available: https://leanpub.com/bddbooks-formulation

[4] "Rspec." [Online]. Available: https://rspec.info/

[5] "Fitness." [Online]. Available: http://fitnesse.org/

[6] "Cucumber." [Online]. Available: https://cucumber.io/docs/guides/overview/

[7] "Specflow." [Online]. Available: https://docs.specflow.org/en/latest/

[8] L. Frantzen, J. Tretmans, and T. Willemse, "A Symbolic Framework for Model-Based Testing," in *FATES/RV'06*, ser. Lecture Notes in Computer Science, K. Havelund, M. Núñez, G. Roşu, and B. Wolff, Eds., vol. 4262. Springer-Verlag, 2006, pp. 40–54. [Online]. Available: https://doi.org/10.1007/11940197_3

[9] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "Test generation based on symbolic specifications," in *Formal Approaches to Software Testing*, J. Grabowski and B. Nielsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–15. [Online]. Available: https://doi.org/10.1007/978-3-540-31848-4_1

[10] P. van den Bos and J. Tretmans, "Coverage-based testing with symbolic transition systems," in *Tests and Proofs*, D. Beyer and C. Keller, Eds. Cham: Springer International Publishing, 2019, pp. 64–82. [Online]. Available: https://doi.org/10.1007/978-3-030-31157-5_5

[11] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *ICSE 2000 the New Millennium*, June 2000, pp. 314–323. [Online]. Available: https://doi.org/10.1145/337180.337217

[12] D. Harel and H. Kugler, "Synthesizing state-based object systems from lsc specifications," in *Implementation and Application of Automata*, S. Yu and A. Păun, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–33. [Online]. Available: https://doi.org/10.1007/3-540-44674-5_1

[13] D. Harel, H. Kugler, and A. Pnueli, *Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 309–324. [Online]. Available: https://doi.org/10.1007/978-3-540-31847-7_18

[14] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, "Test case generation by means of uml sequence diagrams and labeled transition systems," in *2007 IEEE International Conference on Systems, Man and Cybernetics*, Oct 2007, pp. 1292–1297. [Online]. Available: https://doi.org/10.1109/ICSMC.2007.4414060

[15] W. Krenn, R. Schlick, and B. K. Aichernig, "Mapping uml to labeled transition systems for test-case generation," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 186–207. [Online]. Available: https://doi.org/10.1007/978-3-642-17071-3_10

[16] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *Objects, Models, Components, Patterns*, C. A. Furia and S. Nanz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 269–287. [Online]. Available: https://doi.org/10.1007/978-3-642-30561-0_19

[17] M. Irshad, R. Britto, and K. Petersen, "Adapting behavior driven development (bdd) for large-scale software systems," *Journal of Systems and Software*, vol. 177, p. 110944, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000418

[18] M. Diepenbeck, U. Kühne, M. Soeken, and R. Drechsler, "Behaviour driven development for tests and verification," in *Tests and Proofs*, M. Seidl and N. Tillmann, Eds. Cham: Springer International Publishing, 2014, pp. 61–77. [Online]. Available: https://doi.org/10.1007/978-3-319-09099-3_5

[19] J. Xu, Q. Du, and X. Li, "A requirement-based regression test selection technique in behavior-driven development," in *COMPSAC*, July 2021, pp. 1303–1308. [Online]. Available: https://doi.org/10.1109/COMPSAC51774.2021.00182

[20] N. Li, A. Escalona, and T. Kamal, "Skyfire: Model-based testing with cucumber," in *ICST*, April 2016, pp. 393–400. [Online]. Available: https://doi.org/10.1109/ICST.2016.41

[21] M. Alferez, F. Pastore, M. Sabetzadeh, L. Briand, and J.-R. Riccardi, "Bridging the gap between requirements modeling and behavior-driven development," in *MODELS*, Sep. 2019, pp. 239–249. [Online]. Available: https://doi.org/10.1109/MODELS.2019.00008

[22] S. Sivanandan and Y. C. B, "Agile development cycle: Approach to design an effective model based testing with behaviour driven automation framework," in *ADCOM*, Sep. 2014, pp. 22–25. [Online]. Available: https://doi.org/10.1109/ADCOM.2014.7103243

[23] R. Ramler and C. Klammer, "Enhancing acceptance test-driven development with model-based test generation," in *QRS-C*, July 2019, pp. 503–504. [Online]. Available: https://doi.org/10.1109/QRS-C.2019.00096

[24] T. R. Silva, J.-L. Hak, and M. Winckler, "Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development," in *HCSE*, C. Bogdan, J. Gulliksen, S. Sauer, P. Forbrig, M. Winckler, C. Johnson, P. Palanque, R. Bernhaupt, and F. Kis, Eds. Cham: Springer International Publishing, 2016, pp. 86–107. [Online]. Available: https://doi.org/10.1007/978-3-319-44902-9_7

[25] ——, "A behavior-based ontology for supporting automated assessment of interactive systems," in *ICSC*, Jan 2017, pp. 250–257. [Online]. Available: https://doi.org/10.1109/ICSC.2017.73