An Intermediate Language to Integrate Behavior-Driven Development Scenarios and Model-Based Testing

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

Abstract—**We combine Behavior-Driven Development (BDD) and formal Model-Based Testing (MBT), to benefit from the smooth collaboration among stakeholders in BDD, and from automated testing with precise test cases in MBT. However, textual BDD scenarios written in natural language are not sufficient for formal MBT, as they might be ambiguous and lack the required information for testing. In this paper, we define a building block for the integration of BDD and MBT: an intermediate language, IBDD, to fill the gap between textual scenarios and formal models. Furthermore, we formally translate IBDD to BDD Transition Systems, which are formal models for BDD scenarios. Finally, we introduce three approaches for transforming BDDTSs into Symbolic Transition Systems, for which state-of-the-art MBT test generation algorithms are available.**

Index Terms—**Behavior-Driven Development, Model-Based Testing, Symbolic Transition Systems, BDD Transition Systems**

I. Introduction

The complexity and size of software systems is ever increasing. Consequently, testing such systems is increasingly challenging, taking ever more time and effort. Various testing techniques and approaches have been proposed and investigated to tackle these testing challenges, each having its strengths and weaknesses. Two of these approaches are *Behavior-Driven Development* and *Model-Based Testing*. This paper is aimed at combining these two approaches exploiting their complementary strengths.

MBT. Model-Based Testing is a form of black-box testing where test cases are algorithmically generated from a model of the system-under-test (SUT). This model is an abstract and formal description of the behavior of the SUT, and serves as the specification for generating tests. Though MBT, in particular using formal models, has been extensively researched and a reasonable number of commercial and open-source tools exist, there is no widespread use of MBT in industry as yet. The most important factor hindering its adoption is the need to construct, maintain, and understand these formal models. This requires skills and experience in modelling, abstract thinking, and the use of specialized modelling languages, which often are not optimized towards readability and understandability for non-experts such as developers, customers, business developers, and other stakeholders. This complicates communication with these parties and hinders obtaining feedback and validating MBT models, i.e., getting confidence that the model really models what was intended.

BDD. Behavior-Driven Development is an agile approach to software specification and testing, which is popular in the industry nowadays. Unlike MBT, it originates from software engineering practice. Communication and shared understanding of what the software under development shall do, among all stakeholders of the software, is a key goal of BDD [1], [2], [3]. To achieve this, instances of required software behavior are written in structured natural language in so-called BDD *scenarios*, also referred to as *specification by example*. The most popular style to write these scenarios is the *Given-When-Then* style as advocated in the *Gherkin* notation: *Given* a precondition for the required system state, *When* an action is performed on the system, *Then* the specified actions and resulting state are expected.

BDD scenarios can be used as the basis for (among other things) test case derivation. This, however, is not completely automatic. Though tools like *Cucumber* [4] and *SpecFlow* [5] provide support for transforming a BDD scenario into a test case, a developer still has to define how a *Given*-state is reached, how *When*-actions are performed, and how *Then*-results are checked. Moreover, BDD scenarios, being informal, are not always unambiguous, precise, and consistent. This means that a direct, unambiguous transformation from a BDD scenario into an automated, executable test is not possible, but rather depends on the tester's interpretation and effort. This negatively impacts the quality of the derived tests.

BDD for MBT. As follows from the discussion above, BDD and MBT are complementary in their strengths and weaknesses. The goal of our research is therefore to combine their strengths and arrive at a testing approach where people can develop, understand, and communicate using BDD scenarios, which then serve as the source for the MBT models, which in turn enable the use of state-of-the-art MBT test generation algorithms. This overall approach is depicted in Figure 1.

In our previous paper [6] we presented the initial ideas, and experiments with manual transformation from BDD scenarios

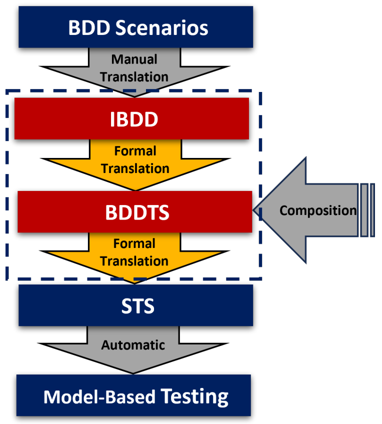


Fig. 1: The flow from BDD Scenarios to MBT. The contribution of this paper lies in the red and yellow part (dashed box).

to a formal MBT model, on which existing test generation techniques are applied [7], [8]. On the one hand, that paper shows that such a manual transformation is feasible. On the other hand, it concludes that the transformation is not easily automated because of the large discrepancy between textual BDD scenarios and formal MBT models. In this paper, we aim to (partially) bridge this gap by defining an *intermediate language*.

IBDD. The intermediate language, coined *IBDD*, follows the structure of the *Given-When-Then* style, but it adds precision by incorporating formal aspects of Symbolic Transition Systems (STS) [7]. We define IBDD in a hybrid grammar, where part of the grammar is in Backus-Naur Form (BNF), but some grammar symbols, which are non-terminals from the BNF point of view, are formal structures appearing in STS, not further elaborated in BNF.

In addition to the grammar, IBDD defines the concept of *Domain-Specific Information* (DSI), which formally captures all relevant information about the domain, for example, type definitions for the global state of the SUT. This type of information is known to developers but is not included in the BDD scenarios themselves. By defining IBDD we make explicit which MBT model elements can be obtained directly from BDD scenarios, which information necessary for making a formal MBT model is missing or implicit, and which aspects of BDD scenarios are ambiguous.

It is important to point out that IBDD is not intended as a language for practitioners to write their scenarios in: the (manual) translation from BDD scenarios to IBDD requires special expertise, as the IBDD syntax is geared towards formal manipulation rather than readability. In that sense, it is only a step towards bridging the gap.

IBDD shall support the transformation from BDD scenarios to formal models, and, in the end, the building of a tool that supports this transformation. Referring to Figure 1, the first step, from BDD scenarios to IBDD, is currently manual. However, in the future, we see possibilities to automate the conversion from natural language BDDs to IBDD, e.g. via natural language processing [9], [10], [11]. The second step transforms the IBDD term into *BDD Transition Systems* (BDDTS), a formal model that is a variant of STS. The next step is the formal transformation of a BDDTS to an STS. This involves extracting the relevant information for the STS from the BDDTS, and adding specific testing information. Finally, existing state-of-the-art algorithms [8] can be used to automatically generate test cases from the STS.

It is worth pointing out that a core benefit of our overall approach is the ability to *compose* smaller BDDTSs into larger ones (gray in Figure 1), along the lines of [6]. However, this is not the topic of the current paper.

This paper presents the definition of IBDD and BDDTS, and the translation from one to the other. We also address several ways to generate STSs from BDDTSs, based on different possibilities for satisfying the precondition (in the *Given*-clause) and checking the postcondition (in the *Then*-clause).

Overview. The next section introduces our running example of an industrial printer. Section III gives some formal background information about mathematical notation, the used algebraic concepts, and formal grammar, all of which are used in the definition of IBDD in Section IV. Section V defines the formal structure of BDDTS, after which Section VI presents the transformation from IBDD to BDDTS. The transformation step from BDDTS to STS is presented in Section VII. Section VIII discusses related work and Section IX summarizes the future work required to cover Figure 1 completely.

II. Motivating Example

We provide 3 real-world scenarios describing an industrial printer to show the applicability of our approach in this paper.

We follow the *Given-When-Then* style to write our scenarios. Each scenario starts with a title describing what the scenario is about, followed by scenario steps.

Scenario 1 : A controller job is added to the scheduled jobs after a job file has been submitted.

Given a job file
When the operator submits the job file using ⟨Submission method⟩
Then the printer appends a new controller job to the scheduled jobs
And the controller job is of type ⟨Job type⟩

In this scenario, Submission method is either LPR, IPP, JMF, or Socket, and Job type is either Production or Streaming.

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

Given a controller job is in the scheduled jobs
And the controller job is a production job
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: At printer restart, the printer controller will clean up printed jobs

Given the printer controller is running
When the operator restarts the printer controller
Then the printed jobs clean up is executed

III. Background

Given two sets A, B, we use $f : A \to B$ (or equivalently $f \in B^A$) to denote a function f from A to B.

A. Algebraic concepts

In the following, we review common algebraic concepts like sorts, functions, variables, terms, and assignments.

Sorts. We assume a universe $\mathcal{S}$ of sorts. In this paper, sorts can be: *(i)* primitive data types (like Bool or Int), *(ii)* composite types $C(d_1 \colon s_1, \ldots, d_n \colon s_n)$, consisting of multiple fields d_i with their own sorts $s_i \in \mathcal{S}$ for $1 \leq i \leq n$, *(iii)* enumerations, whose values are user-defined names, *(iv)* list types $[s]$, whose values are sequences of some other sort $s \in \mathcal{S}$. Examples are the composite type $\mathsf{ControllerJob}(\mathsf{id} \colon \mathsf{Int}, \mathsf{type} \colon \mathsf{JobType}, \mathsf{state} \colon \mathsf{JobState})$, and the enumeration $\mathsf{JobType}$ with values $\mathsf{Production}$, System, $\mathsf{Streaming}$, $\mathsf{ProofPrinting}$.

Functions and terms. We also assume a universe $\mathcal{F}$ of named *functions*, in programming languages often alternatively called methods or procedures. Each function $f \in \mathcal{F}$ has a finite (possibly empty) list of argument sorts $\pi^1 f, \ldots, \pi^{\alpha f} f \in \mathcal{S}$ (where αf is the *arity* of f) and a result sort $\rho f \in \mathcal{S}$, denoted as $f \colon \pi^1 f, \ldots, \pi^{\alpha f} f \to \rho f$. For example, $\mathsf{max} \colon \mathsf{Int}, \mathsf{Int} \to \mathsf{Int}$ is a function with two Int arguments and Int result. A function c with $\alpha c = 0$ is called a *constant*, in which case we write $c : \rho c$; e.g. $\mathsf{true} : \mathsf{Bool}$. Other examples of functions are:

- For every composite type $s = C(d_1 \colon s_1, \ldots, d_n \colon s_n) \in \mathcal{S}$, a constructor $C \colon s_1, \ldots, s_n \to s$ and field selector functions $d_i \colon s \to s_i$ for $1 \leq i \leq n$. Note that d_i applied to $x \in s$ is usually denoted $x.d_i$ rather than $d_i(x)$;
- For every enumeration $s \in \mathcal{S}$, constants $c : s$ for each of the user-defined names of s;
- For every list type $[s]$, functions $\mathsf{is\_empty} : [s] \to \mathsf{Bool}$ to test whether a list is empty, $\mathsf{add} \colon s, [s] \to [s]$ to add an element to the list, and $\mathsf{is\_in\_list} \colon s, [s] \to \mathsf{Bool}$ to test if a certain value is in the list.

Furthermore, we assume a universe of sorted *variables* $\mathcal{V}$. Each $v \in \mathcal{V}$ has an associated sort σv. The value for a variable is not fixed but provided through an *assignment* (see below).

From functions and variables, we build *terms*, which have derived sorts. Essentially, every variable $v \in \mathcal{V}$ is a term of sort σv, and if $f \in \mathcal{F}$ and t_i is a term of sort $\pi^i f$ for $1 \leq i \leq \alpha f$, then $f(t_1, \ldots, t_{\alpha f})$ is a term of sort ρf.[1] We use $t : s$ to denote that t is a term of sort s, and σt to denote that (uniquely determined) sort s. The terms of sort s that contain only variables from $X \subseteq \mathcal{V}$ are collected in $\mathcal{T}_s(X)$. A term t is called *ground* if $t \in \mathcal{T}_{\sigma t}(\emptyset)$. We omit the sort subscript if it is not relevant. We also write $\mathcal{V}_t$ for the set of variables actually occurring in t, i.e., the smallest set such that $t \in \mathcal{T}(\mathcal{V}_t)$

Example terms using a variable $x : \mathsf{Int}$ are: $\mathsf{max(x, 3)}$, and $\mathsf{ControllerJob(x, Production, Printing)}$ (of respective sorts Int and $\mathsf{ControllerJob}(\mathsf{id} : \mathsf{Int}, \mathsf{type} : \mathsf{JobType}, \mathsf{state} : \mathsf{JobState})$).

[1]In addition we also use infix notation and parentheses as usual, for functions that correspond to well-known primitive operators.

Terms of sort Bool are called *predicates*. Examples of predicates are $\mathsf{is\_empty}(x)$ (with $\sigma x = [s]$ for some $s \in \mathcal{S}$) or $x = (5 + y)$ (with variables $x, y : \mathsf{Int}$ and functions $= : \mathsf{Int}, \mathsf{Int} \to \mathsf{Bool}$ and $+ : \mathsf{Int}, \mathsf{Int} \to \mathsf{Int}$).

Assignments. An *assignment* A is a (mathematical) function from variables to terms; to be precise, $A \in \mathcal{T}(Y)^X$ for sets $X, Y \subseteq \mathcal{V}$. For a variable $x \in X$, we say that A assigns term $A(x) \in \mathcal{T}(Y)$ to x. We use $x := t$ to denote the assignment of term t to variable x; e.g., $\mathsf{x} := \mathsf{y} + 1$ denotes $A \in \mathcal{T}(\{y\})^{\{x\}}$ with $A(x) = \mathsf{y} + 1$. As syntactic sugar, if $\sigma x = C(d_1 : s_1, \ldots, d_n : s_n)$ we also write $x.d_i := t$ (with $1 \leq i \leq n$) for $x := C(C.d_1, \ldots, C.d_{i-1}, t, C.d_{i+1}, \ldots, C.d_n)$ (i.e., an assignment that only changes field d_i).

Semantics. The above concepts are all on the level of syntax. In particular, terms are not yet evaluated. For the semantics, we choose the "natural" interpretation of all sorts and functions. This is not further elaborated here.

B. Grammars

We use the Extended BNF (EBNF) to define the IBDD grammar. EBNF is a notation to describe the syntax and structure of formal languages. The basic elements of EBNF notation are given below:

Non-terminals are symbols that represent categories of syntactic elements. They are typically defined using capitalized words enclosed in angle brackets, e.g., $\langle \mathit{Statement} \rangle$.

Terminals are symbols that represent actual tokens or characters. They are typically enclosed in single quotes, e.g., 'if' or '+'. Terminals are the basic building blocks of the language.

Production Rules define how non-terminals can be replaced by a sequence of terminals and/or non-terminals. A production rule consists of a non-terminal on the left side and a sequence of terminals and non-terminals on the right side. Alternative choices can be represented using the "|" symbol.

In addition to these fundamental elements, EBNF grammars can incorporate regular expressions to describe patterns within the language. In particular, the *Kleene Star* $(*)$ denotes zero or more occurrences of a given non-terminal symbol or sub-rule, whereas the *Kleene Plus* $(+)$ denotes one or more occurrences.

IV. Intermediate BDD

In this section, we define the Intermediate Behavior-Driven Development grammar (IBDD). The grammar is a representation of the structural assumptions we make about the BDD scenarios for our modeling purposes. It employs a combination of notations, incorporating elements from both EBNF and mathematical notations derived from first-order logic.

A. Domain Specific Information

The IBDD grammar uses mathematical elements that should be defined for the domain of the BDD scenario, e.g. sort $\mathsf{ControllerJob}$, function $\mathsf{is\_in\_list}$, variable $\mathsf{SJL} : [\mathsf{ControllerJob}]$ and gate $!\mathsf{print}_{\mathsf{start}}$. Some of these elements can be extracted from the textual BDD scenarios, but others are implicit and need to be provided by domain experts.

Definition 1: A DSI is the tuple $(\mathcal{S}, \mathcal{F}, \mathcal{V}, \mathcal{G})$, where

- $\mathcal{S}$ is the set of sort names.
- $\mathcal{F}$ is the set of (well-defined) functions on sorts.
- $\mathcal{V}$ is the set of sorted variables. Variables are categorized into three disjoint sets of global, local, and interaction variables, each explained in subsection IV-B.
- $\mathcal{G}$ is the set of *gates*, standing for interactions between a system and its environment. $\mathcal{G}$ is partitioned into disjoint sets of *input gates* $\mathcal{G}_i$ (initiated by the environment/user, prefixed by '?') and *output gates* $\mathcal{G}_o$ (initiated by the system, prefixed by '!'). Each $g \in \mathcal{G}$ has an associated sequence of sorts $\sigma g \in \mathcal{S}^*$, which determine the kind of values that are communicated through interaction.

B. IBDD Grammar

Given a DSI, we now introduce the EBNF grammar of IBDD. For each production rule, we provide an explanation that also prescribes restrictions on terms where needed. Note that we do not fully specify the syntax; instead, from some point onwards we specify mathematical notation. This serves a dual purpose: streamlining and simplifying the grammar while closely aligning it with the formal model definition.

$$\langle \textit{Scenario} \rangle ::= \langle \textit{Given} \rangle \langle \textit{When} \rangle \langle \textit{Then} \rangle$$

Description:

- The scenario consists of *Given*, *When*, and *Then* clauses.
- *Given*, and *Then* describe the **expectations** from the system while *When* specifies the **behavior** of the system.
- Given an IBDD term, that specifies the sets of local variables LV and interaction variables IV, the set of global variables is derivable by computing $\mathcal{V} \backslash (LV \cup IV)$.

Given clause:

$$\begin{aligned} \langle \textit{Given} \rangle &::= \text{'GIVEN'} \ \langle \textit{Declaration} \rangle \ \text{'['} \ \langle \textit{Guard} \rangle \ \text{']'} \\ \langle \textit{Declaration} \rangle &::= lv_1, \ldots, lv_n \\ \langle \textit{Guard} \rangle &::= P \mid \langle \textit{Guard} \rangle \ \text{'}\wedge\text{'} \ \langle \textit{Guard} \rangle \end{aligned}$$

Description:

- The *Given* clause represents the precondition of the scenario (given by *Guard*) on global and local variables (in which the local variables are given by *Declaration*)
- *Declaration* introduces *local variables* $lv_i \in \mathcal{V}$ for $1 \leq i \leq n$ to the rest of the scenario (*When* and *Then*). Local variables are scoped to the entire scenario.
- *Guard* represents the *input guard* for the scenario. *Guard* is a conjunction (representing the *And* keyword in the textual given step) of predicates P. To have no precondition, i.e. no restriction, choose predicate true for *Guard*.
- $P \in \mathcal{T}_{Bool}(X)$ denotes a predicate on local and global variables $X \subseteq \mathcal{V}$.

When clause:

$$\begin{aligned} \langle \textit{When} \rangle &::= \text{'WHEN'} \ \langle \textit{Switch} \rangle + \\ \langle \textit{Switch} \rangle &::= \langle \textit{Interaction} \rangle \langle \textit{Condition} \rangle \langle \textit{Assignment} \rangle \\ \langle \textit{Interaction} \rangle &::= g.\ iv_1, \ldots, iv_n \\ \langle \textit{Condition} \rangle &::= B \\ \langle \textit{Assignment} \rangle &::= A \end{aligned}$$

Description:

- *When* clause consists of one or more *Switch*es. The Kleene plus (+) in the grammar serves as a representation of the sequential *And* in the textual scenario.
- *Switch* consists of an *Interaction* with the system with corresponding *Condition*s and *Assignment*s.
- *Interaction* consists of a gate $g \in \mathcal{G}$, followed by zero or more distinct *interaction variables* $iv_i \in \mathcal{V}$ for $1 \leq i \leq n$, such that the sorts of the gate and interaction variables match: $\sigma g = \sigma iv_1 \cdots \sigma iv_n$. Interaction variables are scoped to a single switch, i.e. they may only be used in the subsequent *Condition* and *Assignment*.
- *Condition* represents a constraint on variables, with a term $B \in \mathcal{T}_{Bool}(\mathcal{V})$ (which may be true).
- *Assignment* is an assignment function: $A \in \mathcal{T}(\mathcal{V})^X$, where $X \subseteq \mathcal{V}$ includes only local and global variables.

Then clause:

$$\langle \textit{Then} \rangle ::= \text{'THEN'} \ \langle \textit{Switch} \rangle * \ \text{'['} \ \langle \textit{Guard} \rangle \ \text{']'}$$

Description:

- *Then* clause is meant to represent the expected behavior of the scenario, consisting of zero or more output *Switch*es and a post-condition *Guard*.
- *Switch* is the same as defined before in *When*, except that the *Interaction* must have an *output gate* $g \in \mathcal{G}_o$.
- *Guard* defines the post-condition or *output guard*.

V. BDD Transition System

We define a BDD Transition System (BDDTS) as a variant of the Symbolic Transition System (STS) [8].

Definition 2: An STS is a tuple $\langle L, l_0, SV, i, IV, \rightarrow \rangle$, where

- L is a set of locations.
- $l_0 \in L$ is the initial location.
- $SV \subseteq \mathcal{V}$ is a set of state variables containing both global and local variables. They are accessible in the entire transition system.
- $i \in \mathcal{T}(\emptyset)^{SV}$ is the initial assignment of the state variables.
- IV is a set of interaction variables. We assume $SV \cap IV = \emptyset$ and set $V =_{def} SV \cup IV$.
- $\rightarrow \subseteq L \times \mathcal{G} \times IV^* \times \mathcal{T}_{\mathsf{Bool}}(V) \times \mathcal{T}(V)^{SV} \times L$ is the switch relation. In a switch $(l, g, iv_1 ... iv_n, \varphi, a, l') \in \rightarrow$ the elements are called (source) location, gate, interaction variables, guard, assignments, and (destination) location, respectively. The interaction variables $iv_1, \ldots, iv_n$ are distinct, and their sorts are compatible with those of the gate: $\sigma g = \sigma iv_1 \cdots \sigma iv_n$.

A BDDTS adapts the definition of STS by partitioning state variables into local and global ones, adding a goal location and input and output guards. The initial variable assignment is omitted from a BDDTS (it will be added in the transformation from BDDTS to STS, section VII). We note that BDDTS are also defined in [6], as the composition of BDD scenarios; but here, a BDDTS represents a single BDD scenario.

Definition 3: A BDDTS is a tuple $\langle L, l_0, l_G, SV, IV, LV, \rightarrow, IG, OG \rangle$, where

- Elements $L, l_0, SV, IV, \rightarrow$ are as in Definition 2.
- $l_G \in L$ is the *goal location*.

- $LV \subseteq SV$ is the set of *local variables*. Variables $SV \setminus LV$ are called *global*.
- $IG \in \mathcal{T}_{\mathsf{Bool}}(SV)$ is the *input guard* of the BDDTS, denoting the pre-condition of the initial location.
- $OG \in \mathcal{T}_{\mathsf{Bool}}(SV)$ is the *output guard*, denoting the post-condition of the BDDTS in the goal location.

VI. BDD Translation

A. Scenario to IBDD

In this section, we provide an informal translation from textual scenarios to IBDD by example. We construct both the IBDD term and the DSI. The IBDD term for BDD scenario 2 is as follows:

$$\begin{array}{ll}
\mathsf{GIVEN} & \mathsf{CJ}\ [\mathsf{is\_in\_list}(\mathsf{CJ},\mathsf{SJL}) \land \mathsf{CJ.type} = \mathsf{Production}] \\
\mathsf{WHEN} & !\mathsf{print}_{\mathsf{start}}.\mathsf{cj} \\
 & \mathsf{cj.id} = \mathsf{CJ.id} \land \mathsf{cj.state} = \mathsf{Printing} \\
 & \mathsf{CJ.state} := \mathsf{cj.state} \\
 & !\mathsf{print}_{\mathsf{complete}}.\mathsf{cj} \\
 & \mathsf{cj.id} = \mathsf{CJ.id} \land \mathsf{cj.state} = \mathsf{Completed} \\
 & \mathsf{CJ.state} := \mathsf{cj.state}, \mathsf{PJL} := \mathsf{add}(\mathsf{CJ},\mathsf{PJL}) \\
\mathsf{THEN} & [\mathsf{is\_in\_list}(\mathsf{CJ},\mathsf{PJL})]
\end{array}$$

We construct this term as follows. Since scenario 2 starts with '*Given* a controller job', a controller job should be a local variable: CJ. The input guard then constitutes the rest of *Given*. In particular, the guard includes global variables 'scheduled jobs list', and 'printed jobs list', represented as SJL and PJL.

For $\langle$ *When* $\rangle$, we should seek 3 elements: gates with their interaction variables, the conditions of the switches, and assignments. The gates are the actions 'starts printing' and 'completes printing', so we identify respective gates $!\mathsf{print}_{\mathsf{start}}$ and $!\mathsf{print}_{\mathsf{complete}}$. Both actions are applied on 'the controller job', so we identify respective interaction variable cj for this. It is implicit in the BDD scenario what the condition and assignment for the switches with gates $!\mathsf{print}_{\mathsf{start}}$ and $!\mathsf{print}_{\mathsf{complete}}$ should be. For both switches, we expect that the controller job from *Given*, i.e. CJ has the same id as cj, such that CJ is actually being printed. From domain knowledge, we extract that the state of the controller job should change to Printing with gate $!\mathsf{print}_{\mathsf{start}}$ and to Completed with gate $!\mathsf{print}_{\mathsf{complete}}$, so we check this on cj, and update CJ.state accordingly. In the switch of gate $!\mathsf{print}_{\mathsf{complete}}$, we add CJ to PJL to implement the domain knowledge that completing means that the controller job is added to the printed jobs list. Other variables keep their current value (and those assignments are not written explicitly in the IBDD term).

For Scenario 2, the *Then* only contains a statement about the expected state, so the output guard is: $\mathsf{is\_in\_list}(\mathsf{CJ},\mathsf{PJL})$.

Scenarios 1 and 3 have different formats for the *Given*. Scenario 1 only declares a local variable, such that its *Given* clause is just GIVEN JF [true]. Scenario 3 only has an input guard and no local variables: GIVEN [is_running(CTRL)].

Additionally, scenario 1 has a switch in the *Then*. Choosing JT and SM to store the *Job type* and *Submission method* and using getJobType : SubmissionMethod $\rightarrow$ JobType to encode the table, the *Then* clause becomes:

$$\begin{array}{ll}
\mathsf{THEN} & !\mathsf{append}.\ \mathsf{cj},\mathsf{sjl} \\
 & \mathsf{cj.type} = \mathsf{JT} \land \mathsf{sjl} = \mathsf{SJL} \land \mathsf{JT} = \mathsf{getJobType}(\mathsf{SM}) \\
 & \mathsf{CJ} := \mathsf{cj}, \mathsf{SJL} := \mathsf{add}(\mathsf{CJ},\mathsf{SJL}) \\
 & [\mathsf{is\_in\_list}(\mathsf{CJ},\mathsf{SJL}) \land \mathsf{CJ.type} = \mathsf{JT}]
\end{array}$$

The DSI for the printing scenarios can be derived from the three scenarios, except for the set of sort names, and with that the sorts of functions and variables. For example, we should have a sort ControllerJob for variables cj and CJ. Also, the DSI could contain more variables, functions, and gates relevant to the domain.

B. IBDD to BDDTS

This section defines the mapping between the IBDD (using the DSI) and the BDDTS. Below we give an IBDD term that is built from the terminals of the IBDD.

Definition 4: Given the IBDD term:

$$\begin{array}{ll}
\mathsf{GIVEN} & lv_1, \ldots, lv_n\ [IG] \\
\mathsf{WHEN} & g_1.\ iv_{(1,1)}, \ldots, iv_{(1,n_1)}\ B_1\ A_1 \\
 & \vdots \\
 & g_m.\ iv_{(m,1)}, \ldots, iv_{(m,n_m)}\ B_m\ A_m \\
\mathsf{THEN} & g_{m+1}.\ iv_{(m+1,1)}, \ldots, iv_{(m+1,n_{m+1})}\ B_{m+1}\ A_{m+1} \\
 & \vdots \\
 & g_{m+k}.\ iv_{(m+k,1)}, \ldots, iv_{(m+k,n_{m+k})}\ B_{m+k}\ A_{m+k} \\
 & [OG]
\end{array}$$

where numbers $m, k, n, n_i \in \mathbb{N}$ for $1 \leq i \leq m+k$ and $m \geq 1$ are used as indices for the switches of the IBDD term. We define its BDDTS as $\langle\ L, l_0, l_G, SV, IV, LV \rightarrow, IG, OG\ \rangle$ where:

- $L = \{l_0, \ldots, l_m, \ldots, l_{m+k}\}$ is the set of designated locations which we create for index 0 to $m+k$.
- l_0 is the location associated with the first IBDD switch.
- $l_G = l_{m+k}$ is the location of the last switch of the IBDD term, which has associated OG.
- SV is the set of all (local and global) variables occurring in any of the subterms (IG, OG, A_i or B_i for $1 \leq i \leq m+k$), minus the interactions variables IV (defined below).
- $IV = \{iv_{(i,j)} \mid 1 \leq i \leq m+k, 1 \leq j \leq n_i\}$ is the set of all the interaction variables associated with the gates.
- $LV = \{lv_i \mid 1 \leq i \leq n\}$ is the set of all local variables declared in *Given*.
- $\rightarrow = \{(l_{i-1}, g_i, w_i, B_i, A_i, l_i) \mid 1 \leq i \leq m+k\}$, with $w_i = iv_{(1,i)} \cdots iv_{(n_i)}$, is the set of switches from the *When* and *Then*, plus respective source and destination locations from L, to obtain a sequence of switches in the BDDTS.
- IG and OG are as given in the IBDD term.

The BDDTS for scenarios 1,2 and 3 are depicted in Figure 2.

VII. BDDTS TO STS

In this section, we discuss test case generation from individual BDDTSs. This consists of transforming them to STSs and then using techniques from [8] to generate test cases with concrete values for interaction variables. We note once more that our example BDDTS are just sequences of switches. More elaborate (and interesting) cases will be obtained by first composing BDDTSs, as in [6], and then generating test cases from that model. However, the approaches of this section are independent of the complexity of the BDDTS.

In the translation from BDDTS to STS, some choices have to be made. A BDDTS prescribes input and output guards that need to be checked on the state of the system. Furthermore, BDDTSs have local variables whose value is not necessarily uniquely specified by the condition of the *Given* clause. These need to be resolved in testing, and how this is done depends on whether the test interface allows the tester to retrieve the internal state of the system, and in what way. Therefore, we sketch three possible approaches, with different assumptions on this test interface. We explain each approach by example with the STS for the BDDTS of Figure 2b. We note that these three approaches are not an exhaustive list of all possibilities.

We assume that when a BDDTS is translated to an STS, the writer of BDD scenarios can provide values for all global variables, through an initialization function as defined in Definition 2. These values then specify the *expected* values in the scenario (which might be different from the actual values in the system). For the local variables, values are not predetermined but instead established during testing, since they are restricted by the input guard but need not be unique.

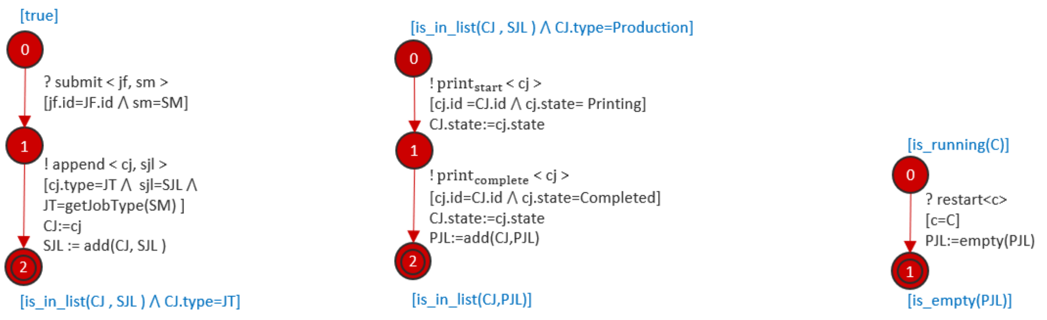
Special gates approach. In this approach, we assume that actual system values of local and global variables can be retrieved from the system via additional switches, that are prepended and appended to the BDDTS. These additional switches use special *check*, *retrieve*, and *choose* gates that are assumed to be gates of the DSI. The input and output guard of the BDDTS are then checked with the values obtained from the system. Figure 3 shows the resulting STS of BDDTS in Figure 2b, where changes w.r.t. the BDDTS are marked bold and green. Its construction is done as follows:

- We prepend a sequence of switches with gates $?\mathsf{check}_{\mathsf{sjl}}$, $!\mathsf{retrieve}_{\mathsf{sjl}}$, and $!\mathsf{choose}_{\mathsf{cj}}$ to the initial location of the BDDTS. With the input switch for $?\mathsf{check}_{\mathsf{sjl}}$, the system is asked to provide its actual value of the scheduled jobs list. Via the output switch $!\mathsf{retrieve}_{\mathsf{sjl}}$, the system then provides this value via interaction variable sjl. This value should be the same as the value of global variable SJL, as expressed by the condition $\mathsf{sjl} = \mathsf{SJL}$.
- Via the output switch with gate $!\mathsf{choose}_{\mathsf{cj}}$, the system provides a controller job value via the interaction variable cj, and hence chooses which controller job is used in the scenario. The value is assigned to the local variable CJ. Unlike *!retrieve*, a *!choose* switch does not need to be preceded by an input switch to request a choice; we assume the system executes $!\mathsf{choose}_{\mathsf{cj}}$ automatically before $!\mathsf{print}_{\mathsf{start}}$. The condition of $!\mathsf{choose}_{\mathsf{cj}}$ is the input guard of the BDDTS, which is now checked on obtained values SJL (equal to sjl of the previous switch) and cj.
- We append switches with gates $?\mathsf{check}_{\mathsf{pjl}}$ and $!\mathsf{retrieve}_{\mathsf{pjl}}$ to the goal location of the BDDTS. The $?\mathsf{check}_{\mathsf{pjl}}$ requests the system to provide the value of the printed jobs list in the $!\mathsf{retrieve}_{\mathsf{pjl}}$ switch. The latter switch also has the output guard as a condition such that it is checked on the retrieved value of pjl with expected controller job CJ. Moreover, the condition of $!\mathsf{retrieve}_{\mathsf{pjl}}$ checks that the provided value of pjl is the same as PJL.

Interaction variables approach. In this approach, we again assume that we can obtain values of variables from the system, but this time the test interface is changed and there are gates available in the DSI with *interaction variables* that can be used to obtain values of local and global variables. Figure 4 shows the resulting STS; again with changes w.r.t. the BDDTS marked bold and green. Its construction is done as follows:

- To retrieve values for the global variables of the input and output guard, respectively, we assume that the switch with gate $!\mathsf{read\&print}_{\mathsf{start}}$ has additional interaction variable sjl and gate $!\mathsf{read\&print}_{\mathsf{complete}}$ has interaction variable pjl.
- The value for the controller job is provided via interaction variable cj in the first switch, and then used to initialize local variable CJ.
- We conjoin $\mathsf{sjl} = \mathsf{SJL}$ to the condition of the first switch to express that the provided value sjl from the system for the scheduled jobs is the same as global variable SJL.
- We conjoin $\mathsf{pjl} = \mathsf{add}(\mathsf{cj}, \mathsf{PJL})$ to the condition of the last switch. This expresses that after the assignment $\mathsf{PJL} := \mathsf{add}(\mathsf{CJ}, \mathsf{PJL})$, the value of pjl provided by the system should equal the global variable PJL.
- We conjoin the input guard with the condition of the first switch, and the output guard with the last switch, where the global variables are substituted by their respective interaction variables. This way the guards are checked on the actual values of the system.

Continued testing approach. In this approach, we do not assume that we can obtain values for global variables from the system via the test interface, as in the other approaches. We only assume that the first switch has interaction variables for obtaining values to initialize local variables because, without a value for local variables, we cannot construct an executable test. Instead of adapting the test interface, we assume that the BDDTS can be extended with additional switches such that in the continuation of the test it can somehow be checked whether the values of the global variables fail to match the state of the system, in which case a guard must have been violated. For example, techniques such as distinguishing sequences could be investigated for this [12]. As a best-effort approach, the input and output guards are conjoined with the conditions of the first and last switch. For any global variable with no corresponding interaction variable, we just use the global variable itself. Hence, if a switch provides no interaction variables to obtain values for global variables, the evaluation of the guard is



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

Fig. 2: BDDTSs translated from IBDD terms in section VI

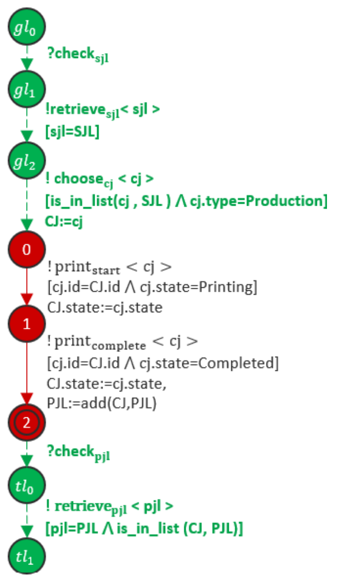


Fig. 3: Scenario 2 test model with special gates

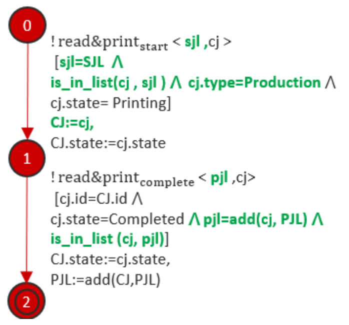

0
! read&print$_{start}$ < sjl ,cj >
[sjl=SJL ∧
is_in_list(cj , sjl) ∧ cj.type=Production ∧
cj.state= Printing]
CJ:=cj,
CJ.state:=cj.state
1
! read&print$_{complete}$ < pjl ,cj>
[cj.id=CJ.id ∧
cj.state=Completed ∧ pjl=add(cj, PJL) ∧
is_in_list (cj, pjl)]
CJ.state:=cj.state,
PJL:=add(CJ,PJL)
2

Fig. 4: Scenario 2 test model with interaction variables

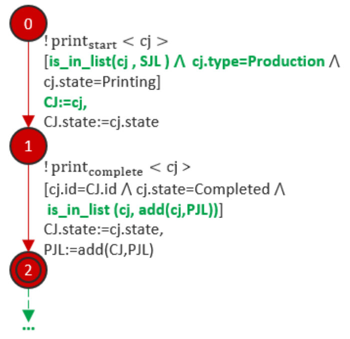

0
! print$_{start}$ < cj >
[is_in_list(cj , SJL) ∧ cj.type=Production ∧
cj.state=Printing]
CJ:=cj,
CJ.state:=cj.state
1
! print$_{complete}$ < cj >
[cj.id=CJ.id ∧ cj.state=Completed ∧
is_in_list (cj, add(cj,PJL))]
CJ.state:=cj.state,
PJL:=add(CJ,PJL)
2
...

Fig. 5: Scenario 2 test model with continued testing

merely a validation of the model itself, and the continuation of the test case is assumed to establish the truth of the guard.

Figure 5 shows the STS constructed with this approach from the BDDTS of Figure 2b. In the STS, the input guard is checked in the condition of the first switch, using the available interaction variable cj and global variable SJL. The output guard is checked in the condition of the last switch. However, because CJ is actually added to PJL in the switch assignment, in the condition we substitute PJL by the right-hand side $\mathsf{add}(\mathsf{CJ}, \mathsf{PJL})$, and (simultaneously) CJ by interaction variable cj that provides the actual value of the controller job.

VIII. Related Work

There have been numerous approaches trying to formalize natural language requirements. Some like [13][14] propose refining informal requirements to formal specification. Some use rigorous requirements languages like in [15]. For UML, several methods have been introduced to make the specifications formal [16] [17]. There is also some work on validating formal models with informal scenarios [18] [19].

BDD scenarios as informal specifications are not sufficient for automatic test generation due to ambiguity and incompleteness [20]. They need to be mapped or translated into a format that makes them suitable for testing. In [21], the authors extend BDD scenarios using annotations to represent the required information for testing and provide a framework based on Cucumber to implement and run tests. I.K.Raharjana et al. [22] developed a tool that generates a mapping table from BDD language to an executable test code in the codeception framework. The previous two approaches do not have a formal translation as we do from IBDD to BDDTS, then STS, and then test case. In [23], they merge testing and formal verification by consolidating test scenarios and formal properties into a single human-readable document.

AI also plays a role in translating scenarios. In [24] they extract the required information from scenarios and generate application code using machine learning models. In [10], they use NLP to semi-automatically extract the code and step definitions from scenarios, and in [25] Neural Network based models are used for metamorphic testing. While AI has its strengths in various natural language processing tasks, careful use, and manual checks are required for making correctness claims, while the formal structure of BNF makes it a reliable choice for the translation of BDD scenarios.

UML-based methodologies for automatic generation of acceptance tests in the form of Gherkin scenarios are provided in [26] and [27]. On the other hand, in [28] and [9], UML models are built from BDD scenarios. However, just like BDD scenarios, UML diagrams might be ambiguous as they lack precise formal semantics. We address this by formalizing the scenarios into IBDD.

IX. Conclusion and Future Work

In this paper, we integrated the BDD and MBT approaches for the automatic generation of test cases. Ambiguity and lack of formal semantics in BDD scenarios and the complexity of formal models in MBT are two of the many reasons behind this integration. Concretely, we defined an intermediate language (IBDD) to fill the gap between textual BDD scenarios and formal BDD Transition Systems. IBDD provides the missing or implicit information required for correctness and precision. IBDD is then translated to BDDTS. We also illustrate three approaches to transform BDDTSs into STSs, from which we can automatically generate test cases [8].

For future work, we plan to complete the process of Figure 1. In particular, we consider: firstly, providing support for translating textual scenarios to IBDD. This step relies on specific prerequisites, such as the necessity for BDD scenarios to adhere to a predefined structure. Secondly, we will investigate how to define different types of BDDTS composition formally. This way we will get complex (i.e., non-sequential) BDDTSs, such that the strength of MBT can be fully exploited for extensive test generation. Thirdly, we plan to provide formal definitions for the translation from BDDTSs to STSs, and investigate which translation approach is applicable in what type of testing context. Finally, we aim to evaluate the whole approach of Figure 1 on a large case study.

Acknowledgment

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.

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

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

[6] T. Zameni, P. van den Bos, J. Tretmans, J. Foederer, and A. Rensink, "From BDD scenarios to test case generation," in *ICSTW*, April 2023, pp. 36–44.

[7] L. Frantzen, J. Tretmans, and T. Willemse, "A Symbolic Framework for Model-Based Testing," in *FATES/RV'06*, ser. LNCS, K. Havelund, M. Núñez, G. Roşu, and B. Wolff, Eds., vol. 4262. Springer-Verlag, 2006, pp. 40–54.

[8] P. van den Bos and J. Tretmans, "Coverage-based testing with symbolic transition systems," in *TAP*, D. Beyer and C. Keller, Eds. Cham: Springer, 2019, pp. 64–82.

[9] A. Gupta, G. Poels, and P. Bera, "Generating multiple conceptual models from behavior-driven development scenarios," *Data & Knowledge Engineering*, vol. 145, p. 102141, 2023.

[10] 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, 2012, pp. 269–287.

[11] J. Fischbach, A. Vogelsang, D. Spies, A. Wehrle, M. Junker, and D. Freudenstein, "Specmate: Automated creation of Test Cases from Acceptance Criteria," in *ICST*. IEEE, 2020, pp. 321–331.

[12] P. van den Bos and F. Vaandrager, "State identification for labeled transition systems with inputs and outputs," *Science of Computer Programming*, vol. 209, p. 102678, 2021.

[13] M. Jastram, S. Hallerstede, M. Leuschel, and A. G. Russo, "An approach of requirements tracing in formal refinement," in *VSTTE*, G. T. Leavens, P. O'Hearn, and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 97–111.

[14] ——, "An approach of requirements tracing in formal refinement," in *Verified Software: Theories, Tools, Experiments*, G. T. Leavens, P. O'Hearn, and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 97–111.

[15] D. ilić, "Deriving formal specifications from informal requirements," in *COMPSAC 2007*, vol. 1, July 2007, pp. 145–152.

[16] M. Giese and R. Heldal, "From informal to formal specifications in UML," in *UML 2004*, T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, Eds. Springer Berlin Heidelberg, 2004, pp. 197–211.

[17] D. Harel, H. Kugler, and A. Pnueli, *Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements*. Berlin, Heidelberg: Springer, 2005, pp. 309–324.

[18] A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra, "A scenario-based validation language for asms," in *Abstract State Machines, B and Z*, E. Börger, M. Butler, J. P. Bowen, and P. Boca, Eds. Berlin, Heidelberg: Springer, 2008, pp. 71–84.

[19] C. Snook, T. S. Hoang, D. Dghaym, and M. Butler, "Domain-specific scenarios for refinement-based methods," in *New Trends in Model and Data Engineering*, C. Attiogbé, F. Ferrarotti, and S. Maabout, Eds. Cham: Springer, 2019, pp. 18–31.

[20] M. Farooq, U. Omer, A. Ramzan, M. A. Rasheed, and Z. Atal, "Behavior driven development: A systematic literature review," *IEEE Access*, vol. 11, pp. 88 008–88 024, 2023.

[21] S. Rodriguez, J. Thangarajah, and M. Winikoff, "A behaviour-driven approach for testing requirements via user and system stories in agent systems," in *AAMAS*. Richland, SC: IFAAMAS, 2023, p. 1182–1190.

[22] I. K. Raharjana, F. Harris, and A. Justitia, "Tool for generating behavior-driven development test-cases," *J. Inf. Syst. Eng. Bus. Intell*, vol. 6, no. 1, p. 27–36, Apr. 2020.

[23] M. Diepenbeck, U. Kühne, M. Soeken, and R. Drechsler, "Behaviour driven development for tests and verification," in *TAP*, M. Seidl and N. Tillmann, Eds. Cham: Springer, 2014, pp. 61–77.

[24] L. Chemnitz, D. Reichenbach, H. Aldebes, M. Naveed, K. Narasimhan, and M. Mezini, "Towards code generation from BDD test case specifications: A vision," in *CAIN*, May 2023, pp. 139–144.

[25] Y. Deng, G. Lou, X. Zheng, T. Zhang, M. Kim, H. Liu, C. Wang, and T. Y. Chen, "BMT: Behavior driven development-based metamorphic testing for autonomous driving models," in *MET*, June 2021, pp. 32–36.

[26] 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.

[27] N. Li, A. Escalona, and T. Kamal, "Skyfire: Model-based testing with Cucumber," in *ICST*, April 2016, pp. 393–400.

[28] I. Lazăr, S. Motogna, and B. Pârv, "Behaviour-driven development of foundational UML components," *Electronic Notes in Theoretical Comp. Science*, vol. 264, no. 1, pp. 91–105, 2010, proc. of FESCA 2010.