JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

PICKLES: a Natural Language Framework for Requirement Specification and Model-Based Testing

María Belén Rodríguez and Petra van den Bos
Formal Methods & Tools, University of Twente, The Netherlands

ABSTRACT

This paper combines methods from the fields of Model-Based Testing (MBT) and Behaviour-Driven Development (BDD) to define a testing approach with human-readable specifications and test cases, as in BDD, while using the modelling techniques and automatic test generation algorithms from MBT. We introduce PICKLES, a Precise Input and Control-flow Keyword-based Language for tEst Scenarios; an extension of Gherkin-style BDD scenarios, specified in structured natural language. We provide a bi-directional translation from Pickles scenarios to formal models, where both specifications and tests are human-readable, and a method to obtain a so-called master model combining all translated scenarios. Standard MBT algorithms can then be applied to automatically derive test cases from it. We implement a prototype of the translation and composition steps, which we use on an industrial case study: a software component from a traffic management system. With it, we illustrate the pipeline and show how Pickles can achieve significantly higher coverage with respect to BDD from the same set of scenarios.

KEYWORDS Behaviour-Driven Development, Model-Based Testing, Symbolic Transition Systems, Domain Specific Language.

1. Introduction

In critical systems, testing plays a central role in ensuring reliability and safety. Exhaustive testing is essential, as it directly strengthens confidence that the system behaves correctly under all expected conditions. Model-Based Testing (MBT) addresses this need by generating test cases from formal models of system behaviour, ensuring broad and systematic coverage.

Despite its advantages, the adoption of MBT in industrial settings is limited. As pointed out by (Alégroth et al. 2022), practitioners recognize that effective MBT adoption depends on the active involvement of all stakeholders in modelling, including those without a technical background. A central barrier then is the difficulty of defining and maintaining formal system models, as expertise in formal specification techniques is required (Villalobos et al. 2019). Furthermore, communicating and discussing such models becomes challenging without a common language, understandable by all parties involved.

In practice, requirements are often specified in natural language (Franch et al. 2023). This is the case for Behaviour-Driven Development (BDD), which has become popular in industry. In BDD, system behaviour is specified through executable examples, typically written as scenarios in a Given-When-Then format. These scenarios serve simultaneously as requirements documentation and as tests, and its human-readable nature facilitates the communication between experts from diverse domains. Nonetheless, example-based specifications can result in large test suites that are costly to maintain (Arredondo Reyes et al. 2024). Moreover, defining requirements in natural language brings key challenges to overcome: ambiguity, inconsistency and incompleteness (Franch et al. 2023).

There is a clear tension in industrial practice: natural language is widely adopted due to its accessibility, however, it suffers from ambiguity and quality issues, while formal MBT models provide precision at the cost of readability and organizational acceptance. Existing approaches force practitioners to choose between these two opposites. To address this gap, we introduce PICKLES, a Precise Input and Control-flow Keyword-based Language for tEst Scenarios. In this framework, both the

JOT reference format:
María Belén Rodríguez and Petra van den Bos. *PICKLES: a Natural Language Framework for Requirement Specification and Model-Based Testing.* Journal of Object Technology. Vol. 25, No. 3, 2026.
http://dx.doi.org/10.5381/jot.2026.25.3.a25

specifications and generated test cases remain human-readable in Pickles format, a Domain Specific Language grounded in BDD-style natural language constructs. A formal model is automatically derived from Pickles specifications and composed into a master model; from it, test cases are generated by means of MBT algorithms, and then translated back to Pickles syntax. This way, we ensure both high-coverage, rigorous testing as well as maintainable, stakeholder-friendly artifacts. The contributions of this paper are as follows:

1. We introduce the PicklesDSL. This domain-specific language preserves the typical Given-When-Then structure used in BDD while introducing constructs for explicitly defining variable domains, constraints, and control flow. This way, Pickles specifications can capture a well-delineated set of requirements, preserving understandability for humans while ensuring unambiguous description of the desired behaviour. The syntax and semantics of this language are introduced in Section 4.
2. We provide a bi-directional translation between PicklesDSL and formal models called Symbolic Transition Systems. Such models are derived *automatically* from the scenarios, and vice versa; thus, test cases can also be human-readable, with the same terminology as the specifications. The translation from PicklesDSL specifications to STSs is introduced in Section 4, whereas Section 6 presents the translation from formal test cases back to PicklesDSL.
3. We define how to compose a so-called *master model* from a set of formal partial models, obtained via translation from a Pickles specification. The master model thus comprises a unified representation of the specified system behaviour, enabling the automatic derivation of tests that better reflect the system's operational behaviour compared to isolated scenarios. This composition is introduced in Section 5.
4. We present a prototype that automates the bi-directional translation, as well as the composition of the master model. This tool is introduced in Section 7.
5. We demonstrate the applicability and advantages of our approach on a case study: a software component for traffic management, from the company Technolution. This case study is introduced in Section 3 and serves as a running example throughout the paper to illustrate the introduced concepts and to discuss the benefits of the approach; this analysis is given in Section 8.

Figure 1 depicts the complete Pickles testing pipeline, including where contributions (1), (2), and (3) take place.

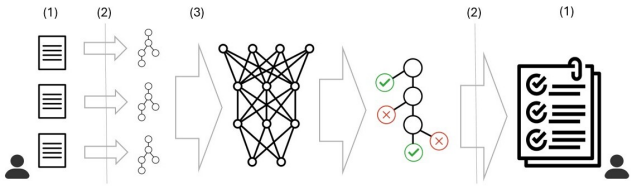


Figure 1 The Pickles testing pipeline: Pickles scenarios (1) are translated (2) into STS, which are then composed (3) into a master model. Formal test cases are derived from the master model, and translated back (2) to Pickles test cases (1).

2. Preliminaries

2.1. Behaviour-Driven Development

Before introducing our case study, we briefly review Behaviour-Driven Development (BDD). BDD is a testing methodology centred around scenarios; concrete, example-based descriptions of how a system is expected to behave under specific conditions. Scenarios are written in Gherkin (*Gherkin* 2025), a domain-specific language that follows a Given–When–Then structure, representing a prescribed situation or context, execution steps of the system, and expected outcomes, respectively. A simple example Gherkin scenario for an ATM system is shown below:

```
Given the user accesses the ATM
And the user has a balance of 100 euros
When the user withdraws 20 euros
Then the user has a balance of 80 euros
```

Scenarios can be transformed into automated test cases by developing concrete implementations for each sentence, often called *step* (or otherwise *keyword*). Tools such as Cucumber (*Cucumber* 2025), Robot Framework (*Robot Framework* 2025), and Behave (*Behave Documentation* 2025) support this process by linking Gherkin specifications to executable test code.

2.2. Variables, types and terms

Across the paper, we reason about variables; their types, domains, and functions over them. With $f : X \to Y$ we denote a total function from a domain X to a codomain Y, with Y^X being the set of functions from X to Y. With T, we denote the universe of variable *types*, where $T_p = \{boolean, integer, decimal, string\}$ is the subset of *primitive types*. We write $\mathcal{T}_\tau(V)$ for a $\tau \in T$ to denote the set of τ-typed terms over variables V, and $\mathcal{T}(V)$ to denote the set of terms over variables V of any type. A term in $\mathcal{T}_\tau(\emptyset)$ is called a *ground term*. A (syntactical) ground term $t \in \mathcal{T}_\tau(\emptyset)$ corresponds to a (semantical) value $u \in \mathcal{U}_\tau$, where $\mathcal{U}_\tau$ denotes the universe of values of type τ. With $\mathcal{U}$ we denote the set of all possible values, of any type.

We also define some auxiliary functions. Function $\mathsf{type}_v : V \to \tau$ returns the *type* (data type) of a variable $v \in V$; $\mathsf{type}_u : \mathcal{U} \to \tau$ does the same for a value $u \in \mathcal{U}$. The function $\mathsf{dom} : V \to \mathcal{P}(\mathcal{U}_\tau)$ yields the *domain* of a given variable, i.e. the set of values it can take; for any $v \in V$, $\mathsf{dom}(v) \subseteq \mathcal{U}_{\mathsf{type}_v(v)}$.

2.3. Symbolic Transition Systems

We will express the semantics of our DSL in terms of Symbolic Transition Systems (STS), as defined in (Van den Bos & Tretmans 2019). This formalism extends Labelled Transition Systems (LTS) with data constructs. In particular, the transition of an STS, extended with data constructs, is called a *switch*. An STS has global variables, called *location variables*, as well as variables local to switches, called *parameters*. Switches consist of an input or output *gate*, and may be augmented with *guards* over parameters and location variables, as well as with *assignments*, that is, updates to the values of location variables.

Definition 1 *A* Symbolic Transition System (STS)*, denoted by* $\mathcal{S}$*, is a tuple* $(\mathcal{L}, l_0, \mathcal{V}_l, \mathcal{V}_p, \Gamma, \mathcal{I}, \mathcal{R})$ *where:*

– $\mathcal{L}$ *is a finite set of locations,*
– $l_0 \in \mathcal{L}$ *is the initial location,*
– $\mathcal{V}_l$ *is a finite set of location variables,*
– $\mathcal{V}_p$ *is a finite set of parameters,*
– $\Gamma = \Gamma_I \cup \Gamma_O$ *is a finite set of input and output gates, with* $\Gamma_I \cap \Gamma_O = \emptyset$,
– *The interaction function* $\mathcal{I} : \Gamma \to \mathcal{V}_p^*$ *maps each* $\gamma \in \Gamma$*, to a sequence* $\bar{p} \in \mathcal{V}_p^*$ *of distinct parameters. Often we use set* $\mathcal{I} \subseteq \Gamma \times \mathcal{V}_p^*$ *instead of function* $\mathcal{I} : \Gamma \to \mathcal{V}_p^*$*, and refer to interaction* $(\gamma, \bar{p}) \in \mathcal{I}$ *instead of writing* $\mathcal{I}(\gamma) = \bar{p}$,
– $\mathcal{R} \subseteq \mathcal{L} \times \mathcal{I} \times \mathcal{T}_{Bool}(\mathcal{V}_l \cup \mathcal{V}_p) \times \mathcal{T}(\mathcal{V}_l \cup \mathcal{V}_p)^{\mathcal{V}_l} \times \mathcal{L}$ *is the switch relation.*

We refer to the elements of a switch $(l, \alpha, \phi, \psi, l') \in \mathcal{R}$, with source location, interaction, guard, assignment, and target location, respectively, and to the elements of interaction $\alpha = (\gamma, p_0 \dots p_k)$, as gate and parameters. We require for each switch that $\phi \in \mathcal{T}_{Bool}(\mathcal{V}_l \cup \{p_0, \dots, p_k\})$, and $\psi \in \mathcal{T}(\mathcal{V}_l \cup \{p_0, \dots, p_k\})^{\mathcal{V}_l}$, i.e. guards and assignments can use any local variable, and only parameters of their switch.

We distinguish input and output switches by their gate, as $\mathcal{R}_I = \{r \in \mathcal{R} \mid \gamma \in \Gamma_I\}$ and $\mathcal{R}_O = \{r \in \mathcal{R} \mid \gamma \in \Gamma_O\}$. Moreover, we use superscripts to refer to an element of a given STS, e.g., $\mathcal{L}^i$ refers to the set of location variables of $\mathcal{S}_i$, and l_0^i to its initial location. In addition, we write $l \xrightarrow{\alpha,\phi,\psi} l'$ to express that $(l, \alpha, \phi, \psi, l') \in \mathcal{R}$, i.e. it is a defined switch, and we write $l \not\xrightarrow{\alpha,\phi,\psi}$ to denote that for any $l' \in \mathcal{L}$ we have $(l, \alpha, \phi, \psi, l') \notin \mathcal{R}$, i.e. the switch does not exist. Note that we do not include a function that assigns initial values to the location variables, as usual for STS (Van den Bos & Tretmans 2019); we define such values when defining test cases in Section 6.1.

2.4. Context-free Grammar

In this work, we describe the Pickles DSL via a context-free grammar P. To this end, we adopt a regex-like variant of Extended Backus–Naur Form (EBNF). In it, non-terminals are enclosed in angle brackets or written in sans-serif font, while terminals have `typewriter font`. We also use the standard symbols "∗", "+", "?", "|", and parentheses in production rules.

Throughout the paper, we denote by s_P the *sub-language* of P generated by a grammar symbol s, that is, the set of all words derivable from s. In addition, we use the notation $[\![w_\mathsf{s}]\!]_\mathsf{s}$ to denote the *semantic interpretation* of w_s, where w_s is a word in s_P. As we define our grammar's semantics by mapping its elements to those of an STS, we may also refer to $[\![w_\mathsf{s}]\!]_\mathsf{s}$ as the *semantic mapping* of the word w_s.

3. Introduction to the Case Study

To test the applicability and benefits of our approach, we apply it to a software component of the Dutch company Technolution B.V. This component is part of a traffic management system; it receives telemetry values from sensors distributed across a road section. These sensors aim to detect abnormal events, such as drivers driving at a low speed or standing still. Based on the events' location, especially with regards to a critical subsection of the road, status information is sent to the user interface.

In Listing 1 we present a specification suite for the system, expressed in PicklesDSL. The suite consists of four scenarios and their domain, expressed in the variable settings prelude. These scenarios were manually derived from a set of BDD test cases, implemented in Cucumber, provided by Technolution. The resulting Pickles scenarios do not correspond one-to-one with the original test cases; instead, during their definition, we incorporated additional domain-knowledge information to generalize the expected behaviour.

```
Variable Settings
"availability" is a string with range {AV, PART AV, NOT AV}
"enabledness" is a boolean with range {true, false}
"faulty detectors" is an array of at most 3
elements where each element is a structure with attributes
"lane", "lenght position" such that:
  "lane" is an integer with range [1,3]
  "length position" is a decimal with range (1.0,3.0)
"critical section lane" is an integer with range [1,3]
"critical section start" is a decimal with range (1.0,3.0)
"critical section end" is a decimal with range (1.0,3.0)

Scenario 01:  faulty detectors outside the critical section
Given the system is in its initial state
And the user interface displays
information on "enabledness", "availability" such that:
  "enabledness" is equal to true AND
  "availability" is equal to AV
When the controller detects "faulty detectors" such that:
  "faulty detectors" has all elements where each element
has attributes such that:
    "lane" is not equal to "critical section lane" OR
    "lenght position" is lower or equal than "critical
section start" OR
    "lenght position" is greater or equal than "critical
section end"
Then the user interface displays "availability" equal to AV

Scenario 02:  one faulty detector in the critical section
Given the system is in its initial state
And the user interface displays
information on "enabledness", "availability" such that:
  "enabledness" is equal to true AND
  "availability" is equal to AV
When the controller detects  "faulty detectors" such that:
  "faulty detectors" has exactly 1 elements where each
element has attributes such that:
    "lane" is equal to "critical section lane" AND
    "lenght position" is greater than "critical section
start" AND
    "lenght position" is lower than "critical section end"
Then the user interface
displays "availability" equal to PART AV

Scenario 03:  at least two faulty detectors in the critical
section
Given the system is in its initial state
And the user interface displays
information on "enabledness", "availability" such that:
  "enabledness" is equal to true AND
  "availability" is equal to AV
When the controller detects "faulty detectors" such that:
  "faulty detectors" has at least 2 elements where each
element has attributes such that:
    "lane" is equal to "critical section lane" AND
    "lenght position" is greater than "critical section
start" AND
    "lenght position" is lower than "critical section end"
Then the user interface
displays "availability" equal to NOT AV

Scenario 04:  lost controller access disables the system
Given the system is in its initial state
And the user interface displays information
on "enabledness" such that:
  "enabledness" is equal to true
When the controller access is lost
Then the user interface reports
status "enabledness" equal to false
```

Listing 1 Pickles Specification Suite for the Technolution case study.

These scenarios define the expected system behaviour when malfunctioning detectors are reported, particularly those located within a critical road section. Each sensor and critical section is localized by means of a discrete lane and a longitudinal position, where the latter represents a continuous value orthogonal to the lanes. At most 3 sensors may fail simultaneously. If there are any number of failing detectors but none of them is in the critical section, the system is available (Scenario 1). If at least one failing sensor is within the critical section, that is, in the same lane and length position, the system reports it is partially available (Scenario 2). If two or more failing detectors match the critical section position, it must report the unavailability of the system (Scenario 3). Finally, if the controller (i.e. whole system) is not accessible at some point, the system is disabled and no other action can be performed afterwards (Scenario 4).

4. Pickles grammar and translation to STS

In this section, we introduce the syntax of the Pickles DSL by means of defining a context-free grammar. We also define its semantics by mapping each of the grammar's elements to elements of a Symbolic Transition System. Throughout this section, we refer to the Pickles scenarios presented in Section 3 to illustrate the introduced concepts.

4.1. Assumptions

While interpretations of the Given–When–Then steps in Gherkin scenarios (see Section 2.1) vary across authors, we constrain ours as follows. The **Given** clause represents the precondition of the scenario, i.e. the state of the System Under Test (SUT) required to execute the scenario. The **When** clause denotes an interaction with the SUT initiated by an external actor, for example, a user. Internal transitions and actions executed by the SUT are excluded. Finally, the **Then** clause represents an observable action performed by the SUT. We assume that all scenarios have at least one When and one Then step.

4.2. Syntax and Semantics

This section presents the concrete mapping from the PicklesDSL grammar, P, to an STS.

As shown in Listing 1, an instance of the PicklesDSL grammar, called *specification suite*, consists of a set of Scenarios and a Variable Settings block. Each scenario describes a functionality in the typical Given–When–Then structure used in BDD; each scenario maps to a STS. As part of this process, each clause preceded by the `Given` token is interpreted as a precondition of the scenario; in the STS, this becomes an additional guard over the first switch. Switches are further defined by `When` and `Then` clauses, which are mapped to input and output switches, respectively. Scenarios may be parametrized using `variables` and guarded by conditions over them. All variables used across scenarios must be declared in the Variable Settings prelude, where their types (e.g. `integer`) and ranges (e.g. `[1, 3]`) are defined. Variables may appear in any step. Conditions over variables can be specified to constrain their admissible values through the use of operators (e.g., `equal`, `between`, `lower than`). Multiple conditions can be connected through logical operators such as `AND` or `OR`. These conditions are mapped to switch guards.

The subsequent subsections define how elements of the PicklesDSL grammar are mapped to STS elements. We first cover the high-level elements before addressing the specific syntax and semantics of the Variable Settings Block, Guard Blocks, and the Given, When, and Then blocks. The corresponding rules for the inverse process, translating formal test cases back into PicklesDSL syntax, will be introduced in Section 6.1.

4.2.1. *Specification Suite*

The non-terminal SpecSuite is the start symbol of our grammar. Its production rule states that a VarSetBlock is followed by one or more Scenarios. Each scenario consists of a textual description (only for readability as it is not taken into account for the mapping to STS), and of Given, When, and Then words, capturing initial conditions, inputs, and expected outputs. These may reference variables, and include guards to specify conditions.

Grammar 1 Pickles Grammar P: Specification Suite

⟨SpecSuite⟩ ::= ⟨VarSetBlock⟩ ⟨Scenario⟩+
⟨VarSetBlock⟩ ::= `Variable Settings` ("⟨VarID⟩" ⟨TypeDesc⟩)*
⟨Scenario⟩ ::= `Scenario` ⟨Str⟩ ⟨Given⟩? ⟨When⟩ ⟨Then⟩
⟨Given⟩ ::= `Given` ⟨InitialCond⟩? ⟨Str⟩? `such that:` ⟨GuardBlock⟩
⟨When⟩ ::= `When` ⟨InStep⟩ (`And` ⟨InStep⟩)*
⟨InStep⟩ ::= ⟨InAction⟩ (("⟨VarID⟩")+ `such that:` ⟨GuardBlock⟩)?
⟨Then⟩ ::= `Then` ⟨OutStep⟩ (`And` ⟨OutStep⟩)*
⟨OutStep⟩ ::= ⟨OutAction⟩ (("⟨VarID⟩")+ `such that:` ⟨GuardBlock⟩)?

Given n words $w_s \in \text{Scenario}_P$ in the SpecSuite, we define a set of n STSs $\mathfrak{S} = \{\mathcal{S}_1, ..., \mathcal{S}_n\}$. Within $\mathfrak{S}$, we distinguish two subsets: primary (or initial) STSs, denoted by $\mathfrak{S}_p$, which correspond to scenarios where the token InitialCond is present; and secondary STSs, denoted by $\mathfrak{S}_s$, which correspond to scenarios where it is absent. Primary STSs can be executed immediately after system initialization, whereas secondary STSs can only be executed following another STS, provided that their guard condition is satisfied. The resulting set is then composed into a single master model, $\mathcal{S}_{master}$; this construction is detailed in Section 5.

4.2.2. *Variable Settings Block*

The VarSetBlock introduces the data types and admissible ranges for all variables in the specification suite. Variables are classified by their data types in primitives (see Section 2.2), arrays, or structures. Grammar 2 shows the production rules for the VarSetBlock non-terminal.

To describe the semantics of the VarSetBlock non-terminal, we first define those of TypeDesc and VarID.

Type Description Words derived from the TypeDesc non-terminal define both the data type and the range for any given variable, which will be mapped to sorts and domains of STS variables. The most fundamental construction is the primitive type, which serves as the building block for more complex definitions. Arrays and structures are constructed through a recursive

Grammar 2 Pickles Grammar P: Variable Settings Block

⟨VarSetBlock⟩ ::= `Variable Settings` ("⟨VarID⟩" ⟨TypeDesc⟩)*
⟨TypeDesc⟩ ::= ⟨Primitive⟩ | ⟨Array⟩ | ⟨Struct⟩
⟨Primitive⟩ ::= `is a` ⟨PrimType⟩ `with range` ⟨Range⟩
⟨PrimType⟩ ::= `boolean` | `string` | `integer` | `decimal`
⟨Array⟩ ::= `is an array of` ⟨Cardinality⟩ `elements where each element` ⟨TypeDesc⟩
⟨Cardinality⟩ ::= `at most` ⟨N⟩ | `exactly` ⟨N⟩ | `between` ⟨N⟩ `and` ⟨N⟩
⟨Struct⟩ ::= `is a structure with attributes` "⟨AttrID⟩" (`,` "⟨AttrID⟩")* `such that:` ⟨AttrDesc⟩+
⟨AttrDesc⟩ ::= "⟨AttrID⟩" ⟨TypeDesc⟩

process, defining the types of their elements or attributes until they eventually resolve into primitive data types. For instance, a structure may be composed of multiple attributes, such as a combination of an integer and an array of strings.

Let $[\![\cdot]\!]_{\mathsf{TypeDesc}} : \mathsf{TypeDesc}_P \to V$ map words in the sub-language $\mathsf{TypeDesc}_P$ to sorted variables. Next, we introduce its definition for primitives, arrays and structures, respectively.

Primitive Definitions For word $w_\mathrm{p} \in \mathsf{Primitive}_P$ of the form:

$$w_\mathrm{p} = \texttt{is a } t \texttt{ with range } r$$

mapping $[\![w_\mathrm{p}]\!]_{\mathsf{Primitive}}$ yields a variable v, with type $[\![t]\!]_{\mathsf{PrimType}}$, and domain $[\![r]\!]_{\mathsf{Range}}$. Here, $[\![\cdot]\!]_{\mathsf{PrimType}} : \mathsf{PrimType}_P \to T_p$ maps each PrimType word to the primitive data type it represents, and $[\![\cdot]\!]_{\mathsf{Range}} : \mathsf{Range}_P \to \mathcal{P}(\mathcal{U}_\tau), \tau \in T_p$ maps range words to sets of primitive values. We do not further define these two functions, and assume that an appropriate representation can be defined for STS, and simply require that $\mathrm{dom}(v) \subseteq \mathcal{U}_{\mathrm{type}(v)}$. For example, the word `is an integer with range {a,b,c}` is not valid as the type and range of the variable are incompatible.

Array Definitions An `array` is a data type that represents a collection of elements of a common type.

Consider a word $w_\mathrm{a} \in \mathsf{Array}_P$ of the form:

$$w_\mathrm{a} = \texttt{is an array of } w_\mathrm{c} \texttt{ elements such that each element } w_\mathrm{e}$$

This word is mapped to an `array` where all elements are of the type given by the word $w_\mathrm{e} \in \mathsf{TypeDesc}_P$. Its domain is the set of all possible arrays of given cardinality w_c, that can be formed with elements in the domain of w_e. For example, the word `is an array of exactly 2 elements such that each element is a string with range {a,b,c}` yields a variable v, which is an array of strings, with as domain the set of all arrays of length 2 consisting of strings `a`, `b` and `c`.

Structure Definitions The `structure` data type denotes a mapping from unique keys to sorted values.

We introduce $[\![\cdot]\!]_{\mathsf{AttrID}}$, a function for mapping words from the AttrID_P sub-language to keys of a structure. There is no constructive definition for this function; instead, we assume that each occurrence of a AttrID consistently refers to the same key, and that this correspondence is recorded for later reuse.

Consider a word $w_\mathrm{s} \in \mathsf{Struct}_P$ of the form:

$$w_\mathrm{s} = \texttt{is a structure with attributes } \texttt{"}w_{\mathrm{att}_0}\texttt{"},\ldots,\texttt{"}w_{\mathrm{att}_n}\texttt{" such that: "}w_{\mathrm{att}_0}\texttt{" } w_{\mathrm{td}_0}\texttt{, ..., "}w_{\mathrm{att}_n}\texttt{" } w_{\mathrm{td}_n}$$

with $w_{\mathrm{att}_0},\ldots,w_{\mathrm{att}_n} \in \mathsf{AttrID}_P$, and $w_{\mathrm{td}_0},\ldots,w_{\mathrm{td}_n} \in \mathsf{TypeDesc}_P$. Then $[\![w_\mathrm{s}]\!]_{\mathsf{Struct}}$ yields a `structure`, where the i-th attribute key is given by w_{att_i} and the i-th attribute's type description is given by w_{td_i}, denoting both its sort and range. For example, the word `is a structure with attributes "AttrA", "AttrB" such that: "AttrA" is an integer with range [1], "AttrB" is a string with range {a,b}` yields a variable of type `structure`, and its domain is the set: $\{\{\texttt{"AttrA"} : 1, \texttt{"AttrB"} : \mathrm{a}\}, \{\texttt{"AttrA"} : 1, \texttt{"AttrB"} : \mathrm{b}\}\}$.

Variable ID In the VarSetBlock, words derived from the VarID non-terminal denote both *location variables* and *gate parameters* in the resulting STS. They are not distinguished in the VarSetBlock; instead, the mapping resolves this when constructing the STS by defining simultaneously a location variable and a gate parameter from the same identifier, both with the same type and domain.

The function $[\![\cdot]\!]_{\mathsf{VarID}} : \mathsf{VarID}_P \to \mathcal{V}_l \times \mathcal{V}_p$ maps words in VarID to tuples of location variable, gate parameter. We also define the projection functions $\pi_l : \mathcal{V}_l \times \mathcal{V}_p \to \mathcal{V}_l$ and $\pi_p : \mathcal{V}_p \times \mathcal{V}_l \to \mathcal{V}_p$. We assume the correspondence between identifications and variables is always consistent and recorded.

Variable Settings Block In the VarSetBlock restrictions for the datatypes and ranges of all variables are introduced. As location variables and gate parameters with the same identifier must be consistent, at all times for an identifier *id* then $\mathsf{type}_v(\pi_l([\![id]\!]_{\mathsf{VarID}})) = \mathsf{type}_v(\pi_p([\![id]\!]_{\mathsf{VarID}}))$ holds. Then, for some word $w_\mathrm{vs} \in \mathsf{VarSetBlock}_P$ of the form:

$$w_\mathrm{vs} = \texttt{Variable Settings "}w_{\mathrm{id}_0}\texttt{" } w_{\mathrm{td}_0} \texttt{ ... "}w_{\mathrm{id}_m}\texttt{" } w_{\mathrm{td}_m}$$

with $w_{\mathrm{td}_0},\ldots,w_{\mathrm{td}_m} \in \mathsf{TypeDescr}_P$ and $w_{\mathrm{id}_0},\ldots,w_{\mathrm{id}_m} \in \mathsf{VarID}_P$, we define:

$$\mathcal{V}_l = \{\pi_l([\![w_{\mathrm{id}_j}]\!]_{\mathsf{VarID}}) \mid 0 \le j \le m\}$$
$$\mathcal{V}_p = \{\pi_p([\![w_{\mathrm{id}_j}]\!]_{\mathsf{VarID}}) \mid 0 \le j \le m\}$$

as the sets of location variables and gate parameters shared by all the STSs derived from the corresponding specification suite. For all $v_j \in \mathcal{V}_l$, $p_j \in \mathcal{V}_p$:

$$\mathsf{type}_v(v_j) = \mathsf{type}_v(p_j) = \mathsf{type}_v([\![w_{\mathrm{td}_j}]\!]_{\mathsf{X}_\mathrm{j}})$$
$$\mathsf{dom}(v_j) = \mathsf{dom}(p_j) = \mathsf{dom}([\![w_{\mathrm{td}_j}]\!]_{\mathsf{X}_\mathrm{j}})$$

where X_j denotes the syntactic category of w_{td_j}: Primitive, Array or Struct, as defined in Grammar 2.

All words derived from VarID used across the Specification Suite must be introduced in VarSetBlock. We also restrict $\mathsf{VarID}_P \cap \mathsf{AttrID}_P = \emptyset$, meaning that variables and structures' attributes may not have identical identifiers.

Example 4.1 *We consider the example introduced in Listing 1, in particular, the definition of the variable* faulty detectors *(see lines 4-6). Table 1 presents a summary of all the variable naming that we will conserve throughout the paper. In addition, we consider* $[\![lane]\!]_{\mathsf{AttrID}} = l$ *and* $[\![length\ position]\!]_{\mathsf{AttrID}} = lp$ *The mapping of* faulty detectors *yields both a location variable,* v_{fd}*, and a gate parameter* p_{fd}*. In both cases, it is an array of one to three structures, each with two attributes:* l*, an integer in* $\{1,2,3\}$*, and* lp*, a decimal strictly between* 1.0 *and* 3.0*.*

Table 1 Variables in Listing 1 and their identifiers.

id	$\pi_l([\![id]\!]_{\mathsf{VarID}})$	$\pi_p([\![id]\!]_{\mathsf{VarID}})$
availability	v_{av}	p_{av}
enabledness	v_{en}	p_{en}
faulty detectors	v_{fd}	p_{fd}
critical section lane	v_{cl}	p_{cl}
critical section start	v_{cs}	p_{cs}
critical section end	v_{ce}	p_{ce}

4.2.3. Guard Block

As mentioned previously, the Pickles specification suite does not define system behaviour as a mere sequence of steps. These steps can be augmented with variables and guarded conditions, enabling parametrization. This shifts the focus from isolated examples to a generalized behavioural description of the system. Input values can be chosen for variables using some test selection strategy, e.g. equivalence partitioning. The same principle applies to outputs: not only can we define a single exact expected value, but we can also specify predicates over variables, to e.g. specify acceptable ranges.

A GuardBlock word defines restrictions for the values of scenarios' variables in each step. A guard is local to a switch; however, it must be compatible with the variable's type and domain, as defined in the variable settings prelude. The production rules for GuardBlock are presented in Grammar 3.

We can now define $[\![\cdot]\!]_{\mathsf{GuardBlock}} : \mathsf{GuardBlock}_P \to \mathcal{T}_{Bool}(\mathcal{V}_p \cup \mathcal{V}_l)$, a function that maps words in $\mathsf{GuardBlock}_P$ to predicates over location variables and gate parameters.

Grammar 3 Pickles Grammar P: Guard Block

⟨GuardBlock⟩ ::= "⟨VarID⟩"⟨Guard⟩(⟨ConjOp⟩"⟨VarID⟩" ⟨Guard⟩)*
⟨Guard⟩ ::= ⟨PrimGuard⟩ | ⟨ArrayGuard⟩ | ⟨StructGuard⟩
⟨PrimGuard⟩ ::= `is` ⟨Op⟩ (⟨ExpValue⟩ | ⟨VarRef⟩)
⟨Op⟩ ::= `equal to` | `greater than` | `lower than` | `lower or equal than` | `greater or equal than`
⟨ArrayGuard⟩ ::= `has` ⟨Quantifier⟩⟨N⟩ `elements where each element` ⟨Guard⟩
⟨Quantifier⟩ ::= `at least` | `at most` | `exactly` | `all`
⟨StructGuard⟩ ::= `has attributes such that:` ⟨AttrGuard⟩ (⟨ConjOp⟩⟨AttrGuard⟩)*
⟨AttrGuard⟩ ::= "⟨AttrID⟩" ⟨Guard⟩
⟨VarRef⟩ ::= `stored?` "⟨VarID⟩"
⟨ConjOp⟩ ::= `AND` | `OR`
⟨ExpValue⟩ ::= ⟨Range⟩ | ⟨N⟩ | ⟨R⟩ | ⟨Str⟩ | ⟨Aexp⟩

Consider a word $w_{\text{gb}} \in \mathsf{GuardBlock}_P$:

$$w_{\text{gb}} = \text{"}w_{\text{id}_0}\text{"}\ w_{\text{g}_0}\ w_{\text{c}_0}\ \ldots\ w_{\text{c}_{n-1}}\ \text{"}w_{\text{id}_n}\text{"}\ w_{\text{g}_n}$$

such that $w_{\text{id}_0} \ldots w_{\text{id}_n} \in \mathsf{VarID}_P$, $w_{\text{c}_0} \ldots w_{\text{c}_{n-1}} \in \mathsf{ConjOp}_P$ and $w_{\text{g}_0} \ldots w_{\text{g}_{n-1}} \in \mathsf{Guard}_P$. $[\![w_{\text{gb}}]\!]_{\mathsf{GuardBlock}}$ yields a predicate over the variables represented by $w_{\text{id}_0} \ldots w_{\text{id}_n}$. Such predicate is built by connecting sub-formulas (each of them potentially over different variables) by means of logic operators, denoted by words $w_{\text{c}_0} \ldots w_{\text{c}_{n-1}}$. Here, $[\![\cdot]\!]_{\mathsf{Conjuntion}} : \mathsf{ConjOp}_P \to \{\wedge, \vee\}$ maps `AND` and `OR` to $\wedge$ and $\vee$ respectively.

We distinguish two cases in the semantics of guards. If a word in GuardBlock is part of a Given word, then the resulting guard restricts the values of location variables, thus, VarID words in it are not interpreted as gate parameters. Conversely, if the GuardBlock occurs in When or Then, each VarID word is interpreted as a parameter unless the token `stored` is prepended to the VarID to mark it explicitly as a location variable.

We now introduce the concrete function definition for a Guard consisting of primitive, array, or structure variables.

Primitive Guard Words in $\mathsf{PrimGuard}_P$ define guards for variables with types in T_p. Consider a word $w_{\text{pg}} \in \mathsf{PrimGuard}_P$ of the form:

$$w_{\text{pg}} = \texttt{is}\ w_{\text{op}}\ w_{\text{rhs}}$$

with $w_{\text{op}} \in \mathsf{Op}_P$ and $w_{\text{v}} \in \mathsf{ExpValue}_P \cup \mathsf{VarRef}_P$. We define $[\![w_{\text{pg}}, x]\!]_{\mathsf{PrimGuard}}$ which yields a predicate obtained by applying the operator $[\![w_{\text{op}}]\!]_{\mathsf{Op}}$ to a left-hand variable x and a right-hand operand. The right-hand operand is either:

- an expected, concrete value $[\![w_{\text{rhs}}]\!]_{\mathsf{ExpValue}}$, if $w_{\text{v}} \in \mathsf{ExpValue}_P$, or
- a variable reference, if $w_{\text{rhs}} \in \mathsf{VarRef}_P$.

With the non-terminal ExpValue we denote a group of generic syntactic categories such as strings, natural or real numbers, ranges or arithmetic expressions. We assume their conventional interpretation and therefore omit further details. The interpretation of the variable reference, and the left-hand variable depends on the context. We distinguish two possible cases: a guard within a Given word, or within When or Then words.

For a PrimGuard within a Given word, $[\![w_{\text{pg}}, x]\!]_{\mathsf{PrimGuard}}$ yields a predicate over the location variable $\pi_l(x)$. If the right-hand side is a variable reference, i.e. $w_{\text{rhs}} \in \mathsf{VarRef}_P$, it is interpreted as a location variable $\pi_l([\![w_{\text{rhs}}]\!]_{\mathsf{VarID}})$. For example, the word `VarA is greater or equal than 1` maps to the predicate $v_a \geq 1$ and the word `VarA is lower than VarB` maps to the predicate $v_a < v_b$, assuming $\pi_l([\![\text{VarA}]\!]_{\mathsf{VarID}}) = v_a$ and $\pi_l([\![\text{VarB}]\!]_{\mathsf{VarID}}) = v_b$.

For a PrimGuard within a When or Then word, $[\![w_{\text{pg}}, x]\!]_{\mathsf{PrimGuard}}$ also yields a predicate, but over the switch parameter resulting from $\pi_p(x)$. If the right-hand side is a variable reference, i.e. $w_{\text{rhs}} \in \mathsf{VarRef}_P$, it is interpreted as a switch parameter unless it is preceded by the token `stored`, which means it is interpreted as a location variable. For example, the word `VarA is greater or equal than 1` now maps to the predicate $p_a \geq 1$. Furthermore, the word `VarA is lower than VarB` maps to the predicate $p_a < p_b$ and the word `VarA is equal to stored VarC` maps to the predicate $p_a = v_c$, where $\pi_p([\![\text{VarA}]\!]_{\mathsf{VarID}}) = p_a$, $\pi_p([\![\text{VarB}]\!]_{\mathsf{VarID}}) = p_b$ and $\pi_l([\![\text{VarC}]\!]_{\mathsf{VarID}}) = v_c$.

Array Guard Words derived from ArrayGuard define predicates for variables of type array by defining conditions that a subset of their elements must meet. Consider a word $w_{\text{ag}} \in \text{ArrayGuard}_P$ of the form:

$$w_{\text{ag}} = \texttt{has}\ w_{\text{q}}\ w_{\text{n}}\ \texttt{elements where each element}\ w_{\text{g}}$$

with $w_{\text{q}} \in \text{Quantifier}_P$, $w_{\text{n}} \in \text{N}_P$ and $w_{\text{g}} \in \text{Guard}_P$. We define the semantic mapping of such word, applied to a variable x of type `array`, as $[\![w_{\text{ag}}, x]\!]_{\text{ArrayGuard}}$ which yields a predicate over a subset of elements of x. The subset is defined by the quantifier $[\![w_{\text{q}}]\!]_{\text{Quantifier}}$ (where w_{q} may be an expression such as `at least`, `exactly`, `at most`) and a number of elements given by $[\![w_{\text{n}}]\!]_{\text{N}}$. The condition each of the elements e in the subset must meet is given by $[\![w_{\text{g}}, e]\!]_{\text{X}}$, where X is the syntactic category of w_{g}: PrimGuard, ArrayGuard or StructGuard. For example, the word `VarA has at least 2 elements where each element has a value greater than 3` is mapped to the predicate $|\{e \in v_a | e > 3\}| \geq 2$. Here, we assume that the word is a guard within a Given, and $\pi_l([\![\text{VarA}]\!]_{\text{VarID}}) = v_a$ is an array.

Structure Guard Consider a word $w_{\text{sg}} \in \text{StructGuard}_P$ of the form:

$$w_{\text{sg}} = \texttt{has attributes such that:}$$
$$\texttt{"}w_{\text{at}_0}\texttt{"}w_{\text{g}_0}w_{\text{c}_0}\ldots w_{\text{c}_{n-1}}\texttt{"}w_{\text{at}_n}\texttt{"}w_{\text{g}_n}$$

The mapping $[\![w_{\text{sg}}, x]\!]_{\text{StructGuard}}$, for variable x of type `struct`, yields a predicate over the attribute values of x. This predicate is built from guards over attributes; the guard over the i-th attribute is given by $[\![w_{\text{g}_i}, [\![at_i]\!]_{\text{AttrID}}]\!]_{\text{X}_i}$, where X is the syntactic category of w_{g_i}, PrimGuard, ArrayGuard or StructGuard; the concrete choice depends on the type of the attribute $[\![at_i]\!]_{\text{AttrID}}$. In guards of STS we may use helper function $\text{get}_K : struct((k_0,\tau_0),\ldots,(k_n,\tau_n)) \times \{k_0,\ldots,k_n\} \rightarrow \mathcal{U}_{\tau_0} \cup \cdots \cup \mathcal{U}_{\tau_n}$ to obtain the value of a given key in a structure. Conditions over attributes may be connected through conjunction operators, given by $[\![w_{\text{conj}_i}]\!]_{\text{Conj}}$. For example, the word `"VarA" has attributes such that: "AttrA" has a value greater than 2 AND "AttrB" has a value equal to abc` maps to the predicate $\text{get}_{AttrA}(v_a) > 2 \wedge \text{get}_{AttrB}(v_a)$, assuming the word is a guard within a Given, and $\pi_l([\![\text{VarA}]\!]_{\text{VarID}}) = v_a$ is a structure with two attributes: an `integer` AttrA and a `string` AttrB.

Example 4.2 (Guard Block as part of a Given word) *We consider again the example introduced in Listing 1, in particular, Scenario 1 (lines 12 to 15). Its precondition, preceded by* `Given`*, determines the input guard of its corresponding STS, that is, $\mathcal{S}_1$. As the word* `the system is in its initial state` *is present in the precondition, then $\mathcal{S}_1$ is considered a primary STS, that is, $\mathcal{S}_1 \in \mathfrak{S}_p$. The word* `the user interface displays information on "enabledness", "availability" such that: "enabledness" has a value equal to true AND "availability" has a value equal to AV` *denotes that the input guard introduces restrictions over the variables* enabledness *and* availability*, connected by means of a logical* and*. Considering the variable mapping introduced in Table 1, this guard can be formally described as $IG^1 \equiv (v_{av} = AV) \wedge v_{en}$.*

Example 4.3 (Guard Block as part of a When/Then word) *We consider again our example introduced in Listing 1. As each scenario has exactly one input and one output switch, we denote their guards by ϕ_i and ϕ_o, respectively. For readability, we define the predicate* $\text{insideCritSection}(d) \equiv (\text{get}_l(d) = v_{\text{cl}}) \wedge (v_{\text{cs}} < \text{get}_{lp}(d) < v_{\text{ce}})$*. This function evaluates to true if the detector, denoted with d, is inside the critical section (i.e. it has a matching lane and it is within the length position range), and evaluates to false otherwise. Then, using Table 1, we define the guards for the input switches as follows:*

$$\phi_i^1 \equiv |\{d \in p_{\text{fd}} \mid \text{insideCritSection}(d)\}| = 0 \wedge IG^1$$
$$\phi_i^2 \equiv |\{d \in p_{\text{fd}} \mid \text{insideCritSection}(d)\}| = 1 \wedge IG^2$$
$$\phi_i^3 \equiv |\{d \in p_{\text{fd}} \mid \text{insideCritSection}(d)\}| \geq 2 \wedge IG^3$$
$$\phi_i^4 \equiv true \wedge IG^4$$

Guards ϕ_i^1 to ϕ_i^3 express the cases where zero, one, and two or more failing detectors are identified inside the critical road section. Simultaneously, the input guards IG enforce that the system must be enabled, and the current report should be "Available". As the input in Scenario 4 has no parameters, no additional guard is present.

We can now define the guards of output switches:

$$\phi_o^1 \equiv p_{\text{av}} = AV \qquad \phi_o^2 \equiv p_{\text{av}} = PART\,AV$$
$$\phi_o^3 \equiv p_{\text{av}} = NOT\,AV \qquad \phi_o^4 \equiv \neg p_{\text{en}}$$

Similarly, guards ϕ_o^1 to ϕ_o^3 represent the values that the system should report for its outputs: available, partially available or not available. Guard ϕ_o^4 expresses the system is disabled.

4.2.4. Given

The Given block states the preconditions of a Scenario, as well as marking the initial state condition. Grammar 4 describes the Given production rules.

Grammar 4 Pickles Grammar P: Given

⟨Given⟩ ::= `Given` ⟨InitialCond⟩? ⟨Str⟩? (`"`⟨VarID⟩`"`)+ `such that:` ⟨GuardBlock⟩
⟨InitialCond⟩ ::= `The system is in its initial state`

The presence of the InitialCond token determines if the corresponding scenario is in the subset $\mathfrak{S}_p$. It is optionally followed by a string with no semantics; its sole purpose is to improve readability. The guard derived from the GuardBlock in Given determines the *initial guard* of the system, denoted with IG, an additional condition over the location variables that the first switch must meet. Then, for some $w_{\text{gi}} \in \text{Given}_P$, containing a word $w_{\text{gb}} \in \text{GuardBlock}_P$, the resulting input guard IG^i of the associated STS $\mathcal{S}_i$ is defined as: $IG^i \equiv [\![w_{\text{gb}}]\!]_{\text{GuardBlock}}$.

If a scenario has no Given word, then $IG \equiv true$.

4.2.5. When and Then

The When non-terminal defines the interaction with the SUT through InStep words, while the Then specifies the observable response as a set of OutStep. Grammar 5 defines the production rules for When and Then.

Grammar 5 Pickles Grammar P: When and Then

⟨When⟩ ::= `When` ⟨InStep⟩ (`And` ⟨InStep⟩)*
⟨InStep⟩ ::= ⟨InAction⟩ (("⟨VarID⟩")+ `such that:` ⟨GuardBlock⟩)?
⟨InAction⟩ ::= ⟨Str⟩
⟨Then⟩ ::= `Then` ⟨OutStep⟩ (`And` ⟨OutStep⟩)*
⟨OutStep⟩ ::= ⟨OutAction⟩ (("⟨VarID⟩")+ `such that:` ⟨GuardBlock⟩)?
⟨OutAction⟩ ::= ⟨Str⟩

Let $[\![\cdot]\!]_{\mathsf{InAction}} : \mathsf{InAction}_P \to \Gamma_i$ and $[\![\cdot]\!]_{\mathsf{OutAction}} : \mathsf{OutAction}_P \to \Gamma_o$ map each input or output action word to the gate they represent. Again, this is a mere correspondence that we assume to be consistent and recorded, therefore we omit a constructive definition of these mappings. Let $[\![\cdot]\!]_{\mathsf{InStep}} : \mathsf{InStep}_P \to \mathcal{R}_I$ and $[\![\cdot]\!]_{\mathsf{OutStep}} : \mathsf{OutStep}_P \to \mathcal{R}_O$ map step words to their corresponding switch.

Consider a scenario with m step words $w_{\text{step}} \in \mathsf{InStep} \cup \mathsf{OutStep}$, of the form:

$$w_{\text{step}} = w_{\text{a}}\text{"}w_{\text{id}_0}\text{"}\ldots\text{"}w_{\text{id}_q}\text{"}\texttt{such that:}w_{\text{gb}}$$

Thus, $[\![w_{\text{step}}]\!]_{\mathsf{X}}$ for the j-th step is defined as follows:

$$[\![w_{\text{step}_j}]\!]_{\mathsf{X}} = (l_{j-1}, (\gamma_j, p_0, \ldots, p_q), \phi_j, \psi_j, l_j), \quad 1 \le j \le m,$$

$$\gamma_j = \begin{cases} [\![a_j]\!]_{\mathsf{InAction}}, & \text{if } \mathsf{X} = \mathsf{InStep} \\ [\![a_j]\!]_{\mathsf{OutAction}}, & \text{if } \mathsf{X} = \mathsf{OutStep} \end{cases}$$

$$\phi_j = \begin{cases} [\![g_j]\!]_{\mathsf{GuardBlock}} \wedge IG, & \text{if } l_{j-1} = l_0, \\ [\![g_j]\!]_{\mathsf{GuardBlock}}, & \text{otherwise} \end{cases}$$

$$(p_0, \ldots, p_q) = \pi_p([\![w_{\text{id}_0}]\!]_{\mathsf{VarID}}) \ldots, \pi_p([\![w_{\text{id}_q}]\!]_{\mathsf{VarID}}),$$

$$\psi_j = \{\, \pi_l([\![w_{\text{id}_0}]\!]_{\mathsf{VarID}}) := \pi_p([\![w_{\text{id}_q}]\!]_{\mathsf{VarID}}) \mid 0 \le k \le q \,\}$$

Given this transformation, each *When* or *Then* step becomes an input or output switch, respectively. These switches have *actions* γ, defined by the textual description of the step, *guards* ϕ, given by guard blocks as introduced in Section 4.2.3, *parameters* $(p_0, \ldots, p_q)$ determined by the variable identifiers used in the step, and, finally, *assignments* ψ; these store the value of the parameters in the corresponding location variable, with the intention of preserving this value for use in subsequent steps.

We assume that interactions are consistent in all steps across all scenarios, i.e. for any interactions $(\gamma_1, \bar{p_1})$ and $(\gamma_2, \bar{p_2})$ resulting from the translation, $\gamma_1 = \gamma_2$ implies that $\bar{p_1} = \bar{p_2}$. As a result, we can construct a set of interactions $\mathcal{I}$, and also gates Γ, as the union of all interactions and gates, respectively, of the obtained switches resulting from the translation of all Pickles scenarios. We also assume that locations of scenarios are disjoint, i.e. for any pair of scenarios $\mathcal{L}^1 \cap \mathcal{L}^2 = \emptyset$. Note that it is easy to check these assumptions automatically.

To condense specifications, if a step's GuardBlock is a simple guard (i.e. without conjunction) over a primitive variable, we allow one-line expressions where VarID is immediately followed by PrimGuard (see 3), skipping the token `such that`; e.g., line 21 in Listing 1. This is merely a syntactic sugar and does not affect the translation.

Example 4.4 *We can now define the partial models derived from our example introduced in Listing 1. As our specification suite has four scenarios, the resulting set of partial models is* $\mathfrak{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4\}$. *We analyze one of them in further detail. Figure 2 shows the STS resulting from Scenario 1. The gate denoted by* faultyDetectors? *represent the input given by the word* the controller detects "faulty detectors" *and* display! *represents the output* the user interface displays "availability".

5. Model Composition

By applying the translation of Section 4 on a specification suite in Pickles, we obtain a set of primary and secondary STSs, representing partial specifications of the system's behaviour. We are now interested in composing these partial specifications into a unified representation of the system's behaviour.

To this end, we first introduce two composition operators that we apply to construct a master model. Choice composition ($\curlywedge$) merges a set of STSs by unifying their initial location, enabling from it any transition that is initially enabled in one of the individual STS. Sequential composition ($\triangleright$) merges the sink locations of a first STS with the initial location of the second.

To construct the master model, we apply choice composition on both the primary STSs, and all STSs, yielding STS $\mathcal{S}_{ini}$ and $\mathcal{S}_{sys}$, respectively. Then, we sequentially compose $\mathcal{S}_{sys}$ with itself, yielding $\mathcal{S}_{sys} \triangleright \mathcal{S}_{sys}$, which denotes that a scenario may be followed by any scenario any number of times, if the guards of subsequent switches are satisfied. Lastly, we sequentially compose $\mathcal{S}_{ini}$ with $\mathcal{S}_{sys} \triangleright \mathcal{S}_{sys}$, yielding $\mathcal{S}_{ini} \triangleright \mathcal{S}_{sys} \triangleright \mathcal{S}_{sys}$, to denote that an initial scenario must be executed before any other. The master model bundles alternatives between scenarios using choice composition, while sequential composition allows chaining multiple scenarios to generate all possible combinations.

Definition 2 (STSs Choice composition) *Let* $\mathcal{S}_1 = (\mathcal{L}^1, l_0^1, \mathcal{V}_l, \mathcal{V}_p, \Gamma, \mathcal{I}, \mathcal{R}^1)$ *and* $\mathcal{S}_2 = (\mathcal{L}^2, l_0^2, \mathcal{V}_l, \mathcal{V}_p, \Gamma, \mathcal{I}, \mathcal{R}^2)$ *be two STSs. Their choice composition is the STS* $\mathcal{S}_1 \curlywedge \mathcal{S}_2 = (\mathcal{L}^{\curlywedge}, l_0^{\curlywedge}, \mathcal{V}_l, \mathcal{V}_p, \Gamma, \mathcal{I}, \mathcal{R}^{\curlywedge})$, *where:*

$$\mathcal{L}^{\curlywedge} = \mathcal{L}^1 \cup \mathcal{L}^2 \cup \{l_0^{\curlywedge}\} \text{ so that } l_0^{\curlywedge} \notin \mathcal{L}^1 \cup \mathcal{L}^2 \text{ is a fresh location}$$

$$\mathcal{R}^{\curlywedge} = (\mathcal{R}^1 \backslash \{(l, \alpha, \phi, \psi, l') \in \mathcal{R}^1 \mid l = l_0^1\}) \cup (\mathcal{R}^2 \backslash \{(l, \alpha, \phi, \psi, l') \in \mathcal{R}^2 \mid l = l_0^2\}) \cup \{(l_0^{\curlywedge}, \alpha, \phi, \psi, l) \mid (l_0, \alpha, \phi, \psi, l) \in \mathcal{R}^1 \cup \mathcal{R}^2 \wedge l_0 \in \{l_0^1, l_0^2\}\}$$

For $n \in \mathbb{N}, n > 2$, *we define* $\curlywedge_{i \in [1,n]} \mathcal{S}_i = \mathcal{S}_1 \curlywedge \mathcal{S}_2 \curlywedge \ldots \mathcal{S}_n$.

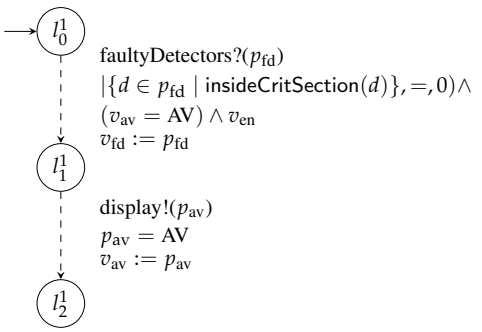


Figure 2 STS_1: Faulty detectors outside the critical section.

Definition 3 (STSs Sequential composition) *Let* $\mathcal{S}_1 = (\mathcal{L}^1, l_0^1, \mathcal{V}_l, \mathcal{V}_p, \mathcal{I}, \Gamma, \mathcal{R}^1)$ *and* $\mathcal{S}_2 = (\mathcal{L}^2, l_0^2, \mathcal{V}_l, \mathcal{V}_p, \mathcal{I}, \Gamma, \mathcal{R}^2)$ *be two STSs, and let* $\mathcal{L}_s^1 = \{l \in \mathcal{L}^1 \mid l \not\xrightarrow{\alpha,\phi,\psi}\}$ *be the set of* sink *locations of* $\mathcal{S}_1$*, that is, the subset of locations without outgoing transitions. We define sequential composition* $\mathcal{S}_1 \triangleright \mathcal{S}_2 = (\mathcal{L}^{\triangleright}, l_0^1, \mathcal{V}_l, \mathcal{V}_p, \Gamma, \mathcal{I}, \mathcal{R}^{\triangleright})$ *where:*

$$\mathcal{L}^{\triangleright} = \mathcal{L}^1 \cup \mathcal{L}^2 \backslash \{l_0^2\}$$
$$\mathcal{R}^{\triangleright} = \mathcal{R}_1 \cup (\mathcal{R}^2 \backslash \{(l, \alpha, \phi, \psi, l') \in \mathcal{R}^2 \mid l = l_0^2\}) \cup$$
$$\{(l^1, \alpha, \phi, \psi, l^2) \mid l^1 \in \mathcal{L}_s^1, l_0^2 \xrightarrow{\alpha,\phi,\psi} l^2\}$$

Definition 4 (Master model of STSs) *Let* $\mathfrak{S}_p$ *and* $\mathfrak{S}_s$ *be two disjoint sets of (primary and secondary) STSs, with* $\mathfrak{S} = \mathfrak{S}_p \cup \mathfrak{S}_s$*. Then we define their* master model *as* $\mathcal{S}_{ini} \triangleright (\mathcal{S}_{sys} \triangleright \mathcal{S}_{sys})$ *where* $\mathcal{S}_{ini} = \curlywedge_{\mathcal{S}_i \in \mathfrak{S}_p} \mathcal{S}_i$ *and* $\mathcal{S}_{sys} = \curlywedge_{\mathcal{S}_i \in \mathfrak{S}} \mathcal{S}_i$*.*

We note that a master model may be non-deterministic. It may also contain unreachable locations due to restrictions enforced by guards; these locations can be discarded after evaluating the satisfiability of the symbolic paths. Consequently, the master model may have unreachable scenarios; this provides feedback to the scenario writer that either the scenarios are wrong, or that such scenarios cannot be tested in the system.

Example 5.1 *Figure 3 presents the master model* $\mathcal{S}_{master}$*, obtained as a result of the synthesis operation* $\curlywedge_{i\in[1,4]} \mathcal{S}_i \triangleright (\curlywedge_{i\in[1,4]} \mathcal{S}_i \triangleright \curlywedge_{i\in[1,4]} \mathcal{S}_i)$*. In this example, we have omitted the paths that cannot be satisfied. For instance, no actions can be performed after executing the switches corresponding to Scenario 4, as after this, the system is no longer enabled, which is a precondition for all scenarios. We note that, in this example, all scenarios are marked as initial. This is because the specification was derived from BDD test cases, which were intended to be executed in isolation, from the initial state of the system, that satisfies some preconditions.*

6. Test Generation and Translation to Pickles

Once a full representation of the SUT is obtained as a Symbolic Transition System applying the methods introduced in sections 4 and 5, it is possible to obtain concrete test cases from it using model-based test generation techniques. We define a formal mapping to express the resulting test cases in Pickles syntax.

6.1. Formal test cases

In this subsection we define formal test cases that can be generated from an STS. We adopt the generation methods proposed in (Van den Bos & Tretmans 2019) to achieve 100% switch coverage; therefore, we focus solely on the format of the resulting test cases and refer the reader to the original work for the underlying generation algorithms. Our formal test cases consist of a sequence of switches in the STS, initial values for location variables, and values for the switch parameters. In (Van den Bos & Tretmans 2019) it is defined how such values can be found via a path condition. This is a Boolean formula that expresses under which conditions all guards along the path evaluate to true, considering the assignments that occur along the way. Values for the variables in the path condition that satisfy the formula can be determined by means of an SMT solver.

Definition 5 (Formal test cases) *Let* $\mathcal{S}$ *be an STS. A* formal test case *is a tuple* $(r_0 \dots r_n, ini, \bar{w}_0 \dots \bar{w}_n)$ *where:*

- $r_0 \dots r_n$ *is a sequence of switches, derived using techniques from (Van den Bos & Tretmans 2019)*
- $ini : \mathcal{V}_l \to \mathcal{U}$ *assigns values to location variables*
- $\bar{w}_0 \dots \bar{w}_n$ *is a sequence of value sequences, such that the values satisfy the path condition for the switches* $r_0 \dots r_n$ *as defined in (Van den Bos & Tretmans 2019).*

6.2. Translation to Pickles

We now cover the mapping from formal test cases to Pickles syntax; in particular, we aim to generate executable test cases compatible with standard BDD tooling. While a formal test case defines values for the parameters of all switches, only those for input switches can be provided to the SUT; outputs are received from it, and may differ from those in the test case. Hence, in this translation, output values are ignored and replaced by the corresponding switch guards to validate the observed behaviour.

To define the translation from test cases to Pickles syntax, we introduce the notation $[\![\cdot]\!]_{\mathsf{X}}^{-1}$; with it, we denote the mapping of mathematical elements to words of the sub-language X_P of the Pickles grammar P. If $[\![\cdot]\!]_{\mathsf{X}}$ was introduced in Section 4, the definition of $[\![\cdot]\!]_{\mathsf{X}}^{-1}$ is omitted as it is the inverse of $[\![\cdot]\!]_{\mathsf{X}}$.

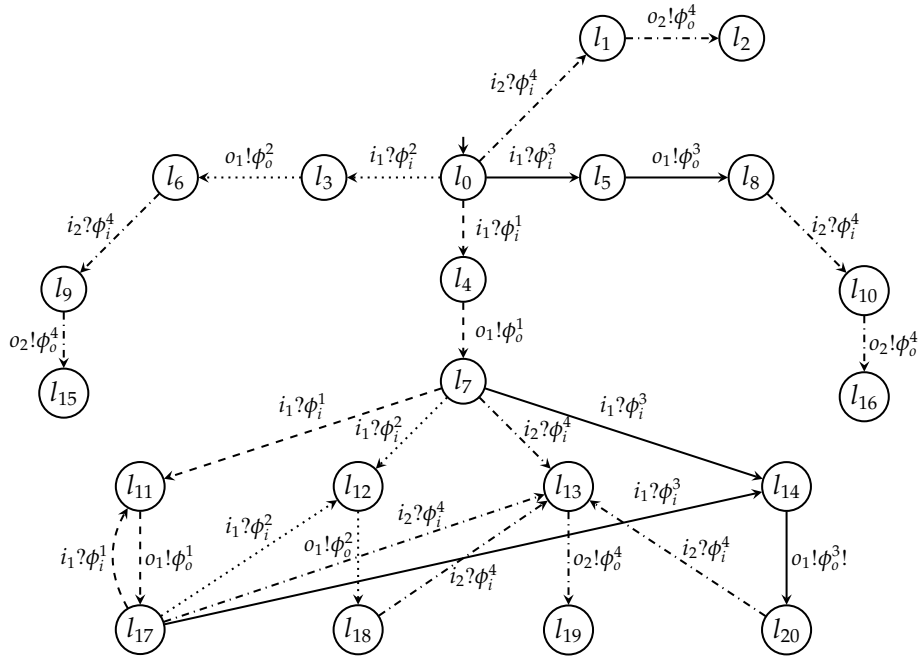
The syntax of a Pickles test case is displayed in Grammar 6. We note that some non-terminals are shared with the specification suite (see Grammar 1); they are reintroduced for clarity.

Consider a formal test case $(r_0 \dots r_n, ini, \bar{w}_0 \dots \bar{w}_n)$, as defined in Definition 5. We first introduce an example of how such test case can be expressed in Pickles syntax.

Example 6.1 *We consider the test case* $(\bar{r}, ini, \bar{w}_0 \dots \bar{w}_n)$*, derived from the model* $\mathcal{S}_{master}$ *presented in Figure 3. Below we define switch sequence* $\bar{r}$*, where subscripts indicate the source and target locations of the each switch. This test starts in the initial location* l_0 *and follows the trace up to location* l_{16}*. Initial values for location variables are defined by ini; their reference can be found in Table 1. Finally,* $\bar{w}_0 \dots \bar{w}_n$ *denotes the parameter value for each switch where, again, subscripts indicate the*

Grammar 6 Pickles Grammar P: Test Case

⟨TestCase⟩ ::= ⟨TestGiven⟩ (⟨TestWhen⟩ | ⟨Then⟩)*
⟨TestGiven⟩ ::= `Given the system is initialized with values:` ⟨ValueDef⟩+
⟨TestWhen⟩ ::= `When` ⟨TestInStep⟩ (`And` ⟨TestInStep⟩)*
⟨TestInStep⟩ ::= ⟨InAction⟩ (("⟨VarID⟩")+ `with values:` ⟨ValueDef⟩+)?
⟨Then⟩ ::= `Then` ⟨OutStep⟩ (`And` ⟨OutStep⟩)*
⟨OutStep⟩ ::= ⟨OutAction⟩ (("⟨VarID⟩")+ `such that:` ⟨GuardBlock⟩)?
⟨ValueDef⟩ ::= "⟨VarID⟩": ⟨Value⟩
⟨Value⟩ ::= ⟨N⟩ | ⟨R⟩ | ⟨Str⟩ | ⟨KeyValue⟩+ | ⟨IndexValue⟩+
⟨IndexValue⟩ ::= ⟨N⟩: ⟨Value⟩
⟨KeyValue⟩ ::= "⟨AttrID⟩":⟨Value⟩



Action	Definition
$i_1?$	`the controller detects "faulty detectors"`
$i_2?$	`the controller access is lost`
$o_1!$	`the interface displays "availability"`
$o_2!$	`the interface reports status "enabledness"`

Guard	Definition
ϕ_i^1	$\lvert\{d \in p_{\text{fd}} \mid \varphi(d)\}\rvert = 0 \wedge (v_{\text{av}} = \text{AV}) \wedge v_{\text{en}}$
ϕ_i^2	$\lvert\{d \in p_{\text{fd}} \mid \varphi(d)\}\rvert = 1 \wedge (v_{\text{av}} = \text{AV}) \wedge v_{\text{en}}$
ϕ_i^3	$\lvert\{d \in p_{\text{fd}} \mid \varphi(d)\}\rvert \geq 2 \wedge (v_{\text{av}} = \text{AV}) \wedge v_{\text{en}}$
ϕ_i^4	v_{en}
ϕ_o^1	$p_{\text{av}} = \text{AV}$
ϕ_o^2	$p_{\text{av}} = \text{PART AV}$
ϕ_o^3	$p_{\text{av}} = \text{NOT AV}$
ϕ_o^4	$\neg p_{\text{en}}$

Figure 3 $\mathcal{S}_{master}$. Each line style (dashed, dotted, solid, dash-dotted) denotes transitions from $\mathcal{S}_1$–$\mathcal{S}_4$. Variables follow the reference in Table 1.

source and target locations, and ϵ denotes an empty sequence.

$\bar{r} = r_{0,5}\, r_{5,8}\, r_{8,10}\, r_{10,16}$

$ini(v_{av}) = AV, \quad ini(v_{en}) = true,$

$ini(v_{cl}) = 1, \quad ini(v_{cs}) = 2.0, \quad ini(v_{ce}) = 2.5$

$ini(v_{fd}) = \{l : 1, lp : 1.5\}, \{l : 1, lp : 1.5\}, \{l : 1, lp : 1.5\}$

$\bar{w}_{0,5} = \{l : 1, lp : 2.0\}, \{l : 1, lp : 2.8\}, \{l : 1, lp : 2.2\}, \bar{w}_{8,10} = \epsilon$

Note that value sequences $\bar{w}$ for output switches are ignored, as we check instead that observed values from the SUT satisfy their guards. Listing 2 presents the test case in Pickles syntax.

```
1  Given the system is initialized with values:
2      "availability": AV
3      "enabledness": true
4      "critical section lane": 1
5      "critical section start": 2.0
6      "critical section end": 2.5
7      "faulty detectors":
8          1:  {"lane":  1, "length position":  1.5}
9          2:  {"lane":  1, "length position":  1.5}
10         3:  {"lane":  1, "length position":  1.5}
11 When the controller detects "faulty detectors" with values:
12     "faulty detectors":
13         1:  {"lane":  1, "length position":  2.0}
14         2:  {"lane":  1, "length position":  2.8}
15         3:  {"lane":  1, "length position":  2.2}
16 Then the user interface
   displays "availability" equal to NOT AV
17 When the controller access is lost
18 Then the user interface reports
   status "enabledness" equal to false
```

Listing 2 Pickles test case executing scenarios 3 and 4 subsequently.

We will now cover the translation details: first, constructing the words of ValueDef (i.e., representations of variables and their values), then, we address the construction of TestGiven, TestWhen and Then words, and finally the complete TestCase.

Value Definition In the following, we assume we have the mapping $[\![\cdot]\!]^{-1}_{\mathsf{Value}} : \mathcal{U} \to \mathsf{Value}_P$. Each word in ValueDef is a concatenation of the textual representation of a variable v with the one of its value u:

$$[\![v, u]\!]^{-1}_{\mathsf{ValueDef}} = [\![v]\!]^{-1}_{\mathsf{VarID}} : [\![u]\!]^{-1}_{\mathsf{Value}}$$

Concretely, for any $u \in \mathcal{U}$, $[\![u]\!]^{-1}_{\mathsf{Value}}$ is defined as follows:

- If $\mathsf{type}_u(u) \in T_p$, then $[\![u]\!]^{-1}_{\mathsf{Value}}$ yields simply the textual representation of the value. For example: $[\![3]\!]^{-1}_{\mathsf{Value}} = 3$.
- If $\mathsf{type}_u(u) = array(\tau)$, then $[\![u]\!]^{-1}_{\mathsf{Value}}$ yields a list with values of each element of u.
- Finally, if $\mathsf{type}_u(u) = struct(K, T)$ then $[\![u]\!]^{-1}_{\mathsf{Value}}$ yields a list with the key-value pairs of u.

Listing 2 provides an example, particularly in lines 7-10: as faulty detectors is an array of structures, its value definition covers each of its three elements, with values for each attribute.

Given In the TestGiven word, in addition to the initial token, a new ValueDef word is created for each location variable $v \in \mathcal{V}_l$, as we need to state the initial values of each. We define $[\![\cdot]\!]^{-1}_{\mathsf{TestGiven}} : \mathcal{V}_l \times (\mathcal{V}_l \to \mathcal{U}) \to \mathsf{TestGiven}_P$ as a mapping from location variables and their initialization function to a word in $\mathsf{TestGiven}_P$. Then, $[\![\mathcal{V}_p, ini]\!]^{-1}_{\mathsf{TestGiven}}$ is defined as:

$$\texttt{Given the system is initialized with values:}\ \{[\![v, ini(v)]\!]^{-1}_{\mathsf{ValueDef}} \mid v \in \mathcal{V}_l\}$$

When For each input switch $r_j \in \mathcal{R}_I$ from the formal test case, a TestInStep within a TestWhen word is produced. A new TestWhen is created if the preceding switch r_{j-1} is an output, otherwise, only a new TestInStep word is appended,

preceded by the token `And`. The InAction is instantiated with the textual representation of the gate γ_j of r_j, i.e., the one stored after parsing the specification suite (see Section 4.2.5). For each gate parameter $p_k \in p_0 \dots p_m$, an entry of ValueBlock is created, with the parameter identifier and its value $u_k \in \bar{w}_j$. Let $[\![\cdot]\!]^{-1}_{\mathsf{TestInStep}} : \mathcal{R}_I \times \mathcal{U}^* \to \mathsf{TestInStep}_P$ map an input switch and a corresponding value sequence to a word. For $r_j = (l_j, (\gamma_j, p_0 \dots p_m), \phi_j, \psi_j, l'_j) \in \mathcal{R}_I$ and a sequence of values $\bar{w} = u_0 \dots u_m$, we define $[\![r_j, \bar{w}]\!]^{-1}_{\mathsf{TestInStep}}$ as:

$$[\![\gamma_j]\!]^{-1}_{\mathsf{InAction}} [\![p_0]\!]^{-1}_{\mathsf{VarID}}, \dots, [\![p_m]\!]^{-1}_{\mathsf{VarID}} \texttt{ with values: } [\![p_0, u_0]\!]^{-1}_{\mathsf{ValueDef}} \dots [\![p_m, u_m]\!]^{-1}_{\mathsf{ValueDef}}$$

Parameter identifiers appear twice in each TestInStep; as a step may reference many parameters, each value in a ValueDef word must be explicitly linked. An example of this mapping can be found in Listing 2, particularly in lines 11 to 15.

Then If r_j is an output switch, i.e. $r_j \in \mathcal{R}_O$, an OutStep within a Then block is produced. If $r_{j-1} \in \mathcal{R}_I$, a new Then block is created. Otherwise, the OutStep word is appended to the existing one, preceded by the token `And`. As noted earlier, output values $\bar{w}_j$ provided by the test case are ignored, as we aim to evaluate the SUT's output values instead. Thus, the textual representation of output switches is recreated as-is from the specification suite; for example, the Then step for Scenario 3 in Listing 1 (line 45) matches the corresponding Then step in Listing 2 (line 16).

Test case A word $w_{\text{tc}} \in$ Test Case can then be defined as the initial Given step defined by $[\![\mathcal{V}_l, ini]\!]^{-1}_{\mathsf{Given}}$, followed by a sequence of n steps corresponding to switches. For the i-th switch, the associated step is prefixed with `And` if the switch $i-1$ has the same type (input or output); otherwise, it is prefixed with `When` for input switches and `Then` for output switches.

7. Implementation

To test our approach, we have implemented a proof-of-concept that: (i) given a set of specification scenarios in Pickles syntax, returns the master STS model and (ii) given this master model and a set of test cases derived from it, returns these tests in Pickles syntax. The implementation was developed in Python, using the Lark library[1], a parsing toolkit with support for ENBF grammar. The tool and the instructions to reproduce the results presented in this paper are publicly available (Rodriguez 2026).

Our tool operates in two modes: specification and test translation. In specification translation mode, it takes as input a plain-text file containing Pickles specifications, such as those introduced in Listing 1. It generates both the definitions of individual STSs and their composition, output in JSON format. In this prototype version of the tool, the master model includes all possible transitions, regardless of path satisfiability. In test translation mode, the tool takes as input an STS in JSON format (including textual descriptions of variables and actions) together with a set of test cases, also represented as JSON and derived from the STS. It outputs the corresponding test cases in plain text Pickles format, such as the one introduced in Listing 2.

[1] https://github.com/lark-parser/lark

8. Discussion

The Pickles approach offers several complementary advantages over traditional BDD-style specification and testing. In this section, we analyse three key characteristics of the framework: (i) human-readable artifacts with well-defined formal semantics, (ii) automatic scenario composition, and (iii) input/output parametrization. To illustrate the advantages related with each, we revisit the example introduced in Section 3 and compare the execution of the four scenarios presented in Listing 1 under two settings: a conventional BDD approach, where scenarios are executed in isolation with concrete values, and Pickles. Note that we do not evaluate the quality of the original test suite provided by Technolution; instead, we show that, with limited additional effort to generalize behaviour and adapt standard BDD scenarios to Pickles, one can obtain the benefits traditionally associated with MBT while maintaining human-readable artifacts.

8.1. Human-readable artifacts with formal semantics

First, by providing a formally defined, deterministic DSL, our framework enforces specifications with precise syntax and semantics suitable for automated testing. Nonetheless, as it is grounded in BDD-style constructs, it also preserves the familiarity and readability of textual specifications. In addition, by abstracting from individual examples to generalized behaviour, Pickles specifications condense information that is typically scattered across requirements, test cases, and auxiliary documents. This improves communication with stakeholders both during requirements definition and on achieved test results.

Equally important is the bi-directional translation between structured natural language and formal models. Human-readable specifications in PicklesDSL constitute a common language across experts with varying technical backgrounds, facilitating precise and collaborative development. At the same time, human-readable tests improve explainability: in many critical systems, it is necessary to provide reports detailing what has been tested and the outcomes of these tests. The recipients of such reports often lack technical expertise, making test scripts in a programming language, or formal models insufficient. Moreover, readable tests create a feedback control loop, enabling the authors of the specifications to review the generated test cases and determine whether the specifications require adjustments.

As our language keeps the Gherkin structure, the resulting test cases can be automated with standard BDD tooling. The implementation of test adapters remains a manual task as in BDD; still, the number of keywords to be automated is always bounded by the number of keywords defined in the specification.

8.2. Automatic scenario composition

Pickles automatically combines individual scenarios into a unified behavioural model. For reactive systems, this yields a more realistic representation than isolated scenarios, as typically specified in BDD, as it captures longer sequences of actions. Such sequences can uncover bugs that emerge from the interaction between successive actions, that would otherwise remain hidden if functionalities were executed in isolation (Zameni et al. 2025). Moreover, scenario composition enables a *shift-left* testing approach: integration, system, and even acceptance-level tests

can be generated and executed at early stages of development, leading to earlier fault detection (Rani et al. 2023).

We illustrate now one of the key advantages of scenario composition: the improvement in transition coverage. Consider Figure 3, showing the master model of the scenarios presented in Listing 1. If the scenarios were executed in isolation, as is typical in BDD, they would cover only 8 of the 26 transitions (about 30%). This limited coverage is visualized in Figure 4, where covered transitions and states are denoted in black, and red represents non-tested transitions. In contrast, our framework, applying the techniques of (Van den Bos & Tretmans 2019) to the same scenarios, yields test cases with 100% transition coverage, an improvement of 70 percentage points. This gain does not imply manual definition of extra scenarios, however, it still needs some additional effort to generalize the system behaviour, including specifying variables and their domains.

From another perspective, consider again the model shown in Figure 3. Achieving 100% transition coverage through manually written test cases would require at least seven tests; the traces from l_0 to l_{15}, l_0 to l_2, l_0 to l_{16} and the four possible traces from l_0 to l_{19}. Pickles achieves the same coverage providing only four scenarios as an input. This represents not only a reduction in manual effort, but also lower maintenance costs: when requirements change, fewer scenarios need to be updated.

8.3. Input/output parametrization

Pickles supports the parametrization of inputs and outputs through explicit variable types and ranges. This enables early consistency checking at the specification level, as ill-typed expressions or incompatible ranges can be detected before any tests are executed. This feature is particularly useful when parts of the specification are produced by different teams, as interfaces can be checked early in the development process.

Moreover, moving from example-based scenarios to more general ones improves input coverage, as an arbitrary number of tests can be generated using any chosen heuristic. For example, consider, in Listing 1, the When step of Scenario 02 (see lines 28-32). The variable "faulty detectors", from now on referred to as v_{fd}, has as domain all arrays of length 1, 2, or 3 containing structures defined by a "lane" (an integer between 1 and 3) and a "length position" (a decimal between 1.0 and 3.0, exclusive). Suppose we generate tests as follows. First, we fix values for the start and end of the critical section to $v_{\text{cs}} = 1.5$ and $v_{\text{ce}} = 2.0$, and its lane to 1, arguing (e.g. based on domain knowledge about our system) that varying these does not provide any gain. Applying boundary analysis, we choose four possible values for the detector's length position attribute: 1.001, 1.5, 2.0 and 2.999. With three lanes and four different length positions, there are 12 possible detector configurations. Note that 2 of these configurations represent detectors in the critical section, i.e. those that satisfy the condition of lines 29-32, and 10 configurations don't. We then have arrays of length 1, 2, or 3 containing exactly one detector configuration in the critical section and zero, one, or two detectors outside the critical section. The total number of valid arrays is then computed as follows, with combinations since order is irrelevant:

$$\binom{2}{1} \cdot \left(\binom{10}{0} + \binom{10}{1} + \binom{10}{2} \right) = 2 \cdot (1 + 10 + 45) = 112$$

By using a single guarded input in the Pickles DSL combined with variable domain definitions, we represent 112 distinct input possibilities. This approach not only reduces the manual effort required to achieve high coverage but, again, also significantly improves test suite maintainability, as changes to the domain automatically propagate to all generated tests.

Extending this reasoning, the input switch in Scenario 01 allows for 175 possible values, while Scenario 03 allows for 11. Scenario 04, having no parameters, sees no benefit. If these scenarios were executed using only one concrete, hardcoded set of values, we would only be exercising approximately 1.5% of the possible system inputs. In BDD, this coverage can be improved by adding more examples; still, this comes at the cost of manual effort and a larger, less maintainable test suite.

The benefits of generalizing the behaviour through input and output ranges are not limited to an improvement in coverage. This approach also reduces tester confirmation bias; when examples are written manually, testers tend to select input values that confirm their expectations about correct behaviour, rather than values that may expose errors (Calikli & Bener 2010).

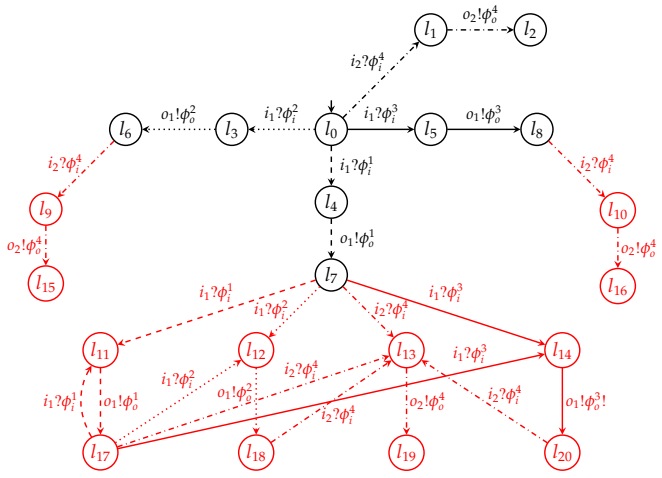


Figure 4 Coverage of STS_{master}. Transitions and states covered by only executing the specification scenarios are shown in black, whereas transitions in red show non-tested transitions.

9. Related Work

The Pickles framework is conceived as a testing framework, but PicklesDSL, as shown throughout this work, is in itself a powerful formalism to unambiguously describe system requirements in a human readable form. Therefore, we review a wide set of related works on the following topics:

1. Natural-language support for requirement specification.
2. Automated test case generation.
3. Natural-language representation of test cases.
4. Parametrization of requirements and test cases.
5. Formal definition of requirements and test cases.

The authors of (Zameni et al. 2024) introduce an intermediate language that translates BDD scenarios into BDD Transition Systems (BDDTS), formal models suitable for automated test generation. This translation needs to be manually conducted;

in this process, additional domain-specific information may be included in the transition systems. In comparison, Pickles condenses all the information needed for unambiguously modelling the system already in a natural language format, with no need for manual translation between readable and formal artifacts. Closely related, the work introduced in (Zameni et al. 2025) defines a sequential composition operator to merge BDDTSs. This is an idea we adopt in our proposal, as part of the model composition introduced in Section 5.

Other authors have covered automatic test generation from structured natural language requirements. In (Carvalho et al. 2014) test generation is automated from controlled natural language specifications in SysReq-CNL. Similarly, authors in (Yue et al. 2015) present RTCM, a restricted natural language for defining Test Case Scenarios from which concrete test cases are automatically derived. Both approaches support requirements parametrization, however, test cases are machine-readable only.

In (Schuts et al. 2025), model-based testing is performed using models generated by ComMA. This tool allows to define a component's behaviour and interfaces in a parametrized and compositional fashion. Similarly, TorXakis (Tretmans 2017) is an MBT tool with support for specifications with complex behaviour such as communication, synchronization, parallelism, non-determinism and data constraints. In both tools, however, neither the specifications nor the test cases are human-readable.

Other approaches generate and translate test cases from (semi-)formal models to human-readable form, i.e. comparable to the last two steps of our pipeline. An MBT pipeline that extracts human-readable test cases from UML diagrams is proposed by (Alsarraj et al. 2025). Additionally, (Alferez et al. 2019) derives Gherkin-style acceptance criteria from UML specifications, generating human-readable tests. Still, both fail to incorporate parametrization, and are aimed specifically for UML users, as specifications are expected to be in this format.

Other work relies on structured natural language for defining formal requirements. Timed requirements and test cases can be expressed with the TEARS framework, presented in (Flemström et al. 2022). In (Autili et al. 2015), a method to specify properties for formal verification by means of Structured English Grammar is introduced. Moreover, authors in (Giannakopoulou et al. 2020) present FRET: a restricted, structured natural language to define requirements that are automatically translated to formally verified temporal-logic formulas. These approaches formally define parametrized system specifications in natural language, but do not provide test generation capabilities.

Finally, Natural Language Processing (NLP)-based testing techniques have grown, particularly with the raise of Large Language Models (Pradel 2025) (Karpurapu et al. 2024) (Rao et al. 2025). However, as noted by (Boukhlif et al. 2024), many studies lack transparency in their evaluation, show narrow benchmarking, and their implementation details are often not disclosed. Still, NLP can play an important role, as noted by (Sudhi et al. 2023): although it is not yet reliable for full automatic formalization from free natural language, it can support formal modelling when applied to structured requirements.

To the best of our knowledge, no existing approach simultaneously covers all topics listed at the beginning of this section.

10. Conclusions and Future Work

This work introduces Pickles, a testing pipeline that bridges the gap between Behavior-Driven Development (BDD) and Model-Based Testing (MBT). By utilizing PicklesDSL, a Gherkin-based language for parametrized specifications and test cases, our approach enables non-technical stakeholders to collaborate on requirements that remain formally rigorous, and consistent. These specifications are automatically synthesized into a formal model, allowing for generating a wide range of test cases. These test cases are expressed back in our DSL, and can then be automated with standard BDD tooling.

As shown through a case study with the company Technolution, using the same number of manually-defined scenarios, with additional, limited effort required to generalize the expected behaviour and migrate to the Pickles format, our framework can increase over 70 percentage points in transition coverage and 98.5 in input coverage compared to BDD. This result illustrates how we leverage MBT advantages while ensuring human-readable artifacts both for requirements and test cases.

For future work, we will expand our implementation and evaluate its performance across a diverse set of case studies, so that we validate our initial analysis of Section 8 at scale. In addition, as the framework's utility relies on its successful adoption by cross-functional teams, we will conduct empirical user studies to systematically evaluate the developer experience, while collecting feedback for further improvement of the tool.

Moreover, we aim to enrich our DSL with a wider range of expressions, e.g. control-flow sequences that allow non-deterministic behaviour. We also plan extensions that allow users to annotate scenarios with information about their criticality and probability of failure; this information can be integrated to the testing algorithm to help identify high-risk bugs faster. Given that timing is a primary constraint in critical systems, we also foresee incorporating time-constraint definitions.

Finally, we plan to investigate ways to assist the migration from BDD to our framework; in particular, how can pre-existing Gherkin test cases be processed, for example by leveraging large-language models, to automatically generalize the expected behaviour of the system from these examples.

Acknowledgments

The research is carried out as part of the NWO project *Evidence-driven Black-box Checking (EVI)*, with project number OCENW.M.23.155.

References

Alégroth, E., Karl, K., Rosshagen, H., Helmfridsson, T., & Olsson, N. (2022). Practitioners' best practices to Adopt, Use or Abandon Model-based Testing with Graphical models for Software-intensive Systems. *Empirical Software Engineering*, *27*(5). doi: 10.1007/s10664-022-10145-2

Alferez, M., Pastore, F., Sabetzadeh, M., Briand, L., & Riccardi, J.-R. (2019). Bridging the Gap between Requirements Modeling and Behavior-Driven Development. In *MODELS'19* (p. 239-249). doi: 10.1109/MODELS.2019.00008

Alsarraj, R. G., Altaie, A. M., & Zuhair Majeed, E. (2025). Developing an Automated Model-Based Software Testing Tool From the Design Phase. *IEEE Access*, *13*, 58548-58558. doi: 10.1109/ACCESS.2025.3553967

Arredondo Reyes, V. M., Domínguez Isidro, S., Sańchez García, Á. J., & Ocharán Hernández, J. O. (2024). Analysis of Behavior-Driven Development: A Thematic Synthesis. *Programming and Computer Software*, *50*(8), 701–713. doi: 10.1134/S0361768824700713

Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., & Tang, A. (2015). Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Transactions on Software Engineering*, *41*(7), 620-638. doi: 10.1109/TSE.2015.2398877

Behave Documentation. (2025). Retrieved from https://behave.readthedocs.io/en/latest/ (Accessed: 2025-09-22)

Boukhlif, M., Hanine, M., Kharmoum, N., Ruigómez Noriega, A., García Obeso, D., & Ashraf, I. (2024). Natural Language Processing-Based Software Testing: A Systematic Literature Review. *IEEE Access*, *12*, 79383-79400. doi: 10.1109/ACCESS.2024.3407753

Calikli, G., & Bener, A. (2010). Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *PROMISE 2010.* doi: 10.1145/1868328.1868344

Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., & Blackburn, M. (2014). NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming*, *95*, 275-297. doi: 10.1016/j.scico.2014.06.007

Cucumber. (2025). Retrieved from https://cucumber.io/ (Accessed: 2025-09-22)

Flemström, D., Afzal, W., & Enoiu, E. P. (2022). Specification of Passive Test Cases Using an Improved T-EARS Language. In *Software Quality: The Next Big Thing in Software Engineering and Quality* (pp. 63–83). Springer. doi: 10.1007/978-3-031-04115-0_5

Franch, X., Palomares, C., Quer, C., Chatzipetrou, P., & Gorschek, T. (2023). The state-of-practice in requirements specification: an extended interview study at 12 companies. *Requirements Engineering*, *28*(3), 377–409. doi: 10.1007/s00766-023-00399-7

Gherkin. (2025). Retrieved from https://cucumber.io/docs/gherkin (Accessed: 2025-09-22)

Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., & Shi, N. (2020). Formal Requirements Elicitation with FRET. In *REFSQ Workshops.* Retrieved from https://api.semanticscholar.org/CorpusID:214708107

Karpurapu, S., Myneni, S., Nettur, U., Gajja, L. S., Burke, D., Stiehm, T., & Payne, J. (2024). Comprehensive Evaluation and Insights Into the Use of Large Language Models in the Automation of Behavior-Driven Development Acceptance Test Formulation. *IEEE Access*, *12*, 58715-58721. doi: 10.1109/ACCESS.2024.3391815

Pradel, M. (2025). *Testora: Using Natural Language Intent to Detect Behavioral Regressions.* (Accepted at ICSE 2026) doi: 10.48550/arXiv.2503.18597

Rani, V. S., Babu, D. A. R., Deepthi, K., & Reddy, V. R. (2023). Shift-Left Testing in DevOps: A Study of Benefits, Challenges, and Best Practices. In *ICACRS'23* (p. 1675-1680). doi: 10.1109/{ICACRS}58579.2023.10404436

Rao, N., Gilbert, E., Green, H., Ramananandro, T., Swamy, N., Le Goues, C., & Fakhoury, S. (2025). *DiffSpec: Differential Testing with LLMs using Natural Language Specifications and Code Artifacts.* doi: 10.48550/arXiv.2410.04249

Robot Framework. (2025). Retrieved from https://robotframework.org/ (Accessed: 2025-09-22)

Rodriguez, M. B. (2026). *PICKLES: a Natural Language Framework for Requirement Specification and Model-Based Testing.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.19710432 doi: 10.5281/zenodo.19710432

Schuts, M., Hoomann, J., Kurtev, I., Tlili, I., & Oerlemans, E. (2025). Online Model-Based Testing Reusing Multiple Design Models in an Industrial Setting. *JOT*, *24*(2), 2:1-14. doi: 10.5381/jot.2025.24.2.a6

Sudhi, V., Kutty, L., & Gröpler, R. (2023). Natural Language Processing for Requirements Formalization: How to Derive New Approaches? In *CS&P'21* (pp. 1–27). Springer. doi: 10.1007/978-3-031-26651-5_1

Tretmans, J. (2017). On the existence of practical testers. In *Lect. Notes Comput. Sci.* (pp. 87–106). Springer. doi: 10.1007/978-3-319-68270-9_5

Van den Bos, P., & Tretmans, J. (2019). Coverage-Based Testing with Symbolic Transition Systems. In *TAP* (pp. 64–82). Springer. doi: 10.1007/978-3-030-31157-5_5

Villalobos, L., Quesada López, C., Martinez, A., & Jenkins, M. (2019). Model-based testing areas, tools and challenges: A tertiary study. *CLEI*, *22*. doi: 10.19153/cleiej.22.1.3

Yue, T., Ali, S., & Zhang, M. (2015). RTCM: a natural language based, automated, and practical test case generation framework. In *ISSTA'15* (p. 397–408). ACM. doi: 10.1145/2771783.2771799

Zameni, T., Van den Bos, P., Foederer, J., & Rensink, A. (2025). Sequential Composition of BDD Transition Systems for Model-Based Testing. In *FORTE'25* (pp. 36–54). Springer. doi: 10.1007/978-3-031-95497-9_3

Zameni, T., Van den Bos, P., Rensink, A., & Tretmans, J. (2024). An Intermediate Language to Integrate Behavior-Driven Development Scenarios and Model-Based Testing. In *SANER-C'24* (pp. 199–206). doi: 10.1109/SANER-C62648.2024.00033

About the authors

María Belén Rodríguez is a PhD Candidate at the University of Twente (The Netherlands). Her research focuses on formal testing and its integration in industrial settings. You can contact the author at mariabelen.rodriguez@utwente.nl.

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