PICKLES: a Natural Language Framework for Requirement Specification and Model-Based Testing (Extended Abstract)

María Belén Rodríguez
Formal Methods & Tools
University of Twente
Enschede, The Netherlands
mariabelen.rodriguez@utwente.nl

Petra van den Bos
Formal Methods & Tools
University of Twente
Enschede, The Netherlands
p.vandenbos@utwente.nl

We combine 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 BDD-style scenarios, specified in structured natural language. We provide a translation from Pickles scenarios to formal models, and a method to obtain a so-called master model that combines all translated scenarios. Standard MBT algorithms can then be applied to automatically derive test cases from it. Our translation is bi-directional, so that we can obtain test cases in the Pickles format again, which can then be executed using standard BDD tooling.

1 Introduction

In critical systems, such as spacecrafts, testing plays a central role in ensuring reliability and safety. Space systems are typically developed under strict reliability constraints, operate in environments where failures are extremely costly or irreversible, and must behave correctly over long mission lifetimes with limited opportunities for intervention. As a result, exhaustive testing is essential, as it directly strengthens confidence that the system behaves correctly under all expected conditions.

Model-Based Testing (MBT) can address this need by generating test cases from formal models of system behaviour, ensuring broad and systematic coverage. Despite its potential, the adoption of MBT in industrial settings is limited. A central barrier is the difficulty of defining and maintaining formal system models, as expertise in formal specification techniques is required [17]. As pointed out by the authors of [3], practitioners recognise that effective MBT adoption depends on the active involvement of all stakeholders in modelling, including those without a technical background. This challenge is particularly pronounced in the space sector, where requirements are specified by experts from highly diverse domains, each with their own terminology and perspective. Establishing a common language that allows these actors to jointly reason about system behaviour is therefore crucial.

In practice, requirements are often specified in natural language [10]. This is the case in Behaviour-Driven Development (BDD), which has become increasingly popular in industry. In BDD, system behaviour is specified through examples, typically written as scenarios in a Given-When-Then format, that can become automated test cases through tools like Cucumber [1] or RobotFramework [2]. These examples, usually called scenarios, serve simultaneously as documentation of requirement and as (executable) test cases. The human-readable nature of BDD scenarios facilitates communication between experts from diverse domains. In particular, space software and firmware development is usually conducted under certified standards, where clearly documenting both requirements and the corresponding tests for verifying them is essential for certification and review. Nonetheless, example-based specifications can

Submitted to:
Models in Space Systems: Integration, Operation, and Networking Workshop 2026
© M.B. Rodríguez & P. van den Bos

result in large test suites that are costly to maintain [14]. Moreover, when defining requirements in natural language, there is a key challenge to overcome: ambiguity, inconsistency and incompleteness [10].

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 understandability and organizational acceptance. To address this gap, we introduce PICKLES, a Precise Input and Control-flow Keyword-based Language for tEst Scenarios. In it, both the specifications and the generated test cases remain human-readable in Pickles format, a natural language Domain Specific Language grounded in BDD-style constructs. A formal model is automatically derived from each of the Pickles specifications and all of them are 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 can ensure both high-coverage, rigorous testing as well as maintainable, stakeholder-friendly artifacts.

2 The Pickles framework

The envisioned contributions of this work are as follows:

1. We present our domain-specific language (DSL) PicklesDSL. It preserves BDD's Given-When-Then structure while introducing constructs for explicitly defining variable ranges, constraints, and control flow. This way, Pickles scenarios can capture a well-delineated set of requirements, preserving understandability for humans while ensuring unambiguous description of the desired behaviour.
2. We provide a bi-directional translation between Pickles scenarios and formal models called Symbolic Transition Systems. Such models can be derived *automatically* from the scenarios, and vice versa; thus, test cases are also human-readable, with the same terminology as the specifications.
3. We define how to compose a so-called *master model* from a set of formal partial models, each of them 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.

Figure 1 shows the Pickles testing pipeline, including where contributions (1), (2), and (3) take place. This paper introduces a high-level description of the approach. Its complete description, including formal definitions and analysis with a real case study, can be found in [12].

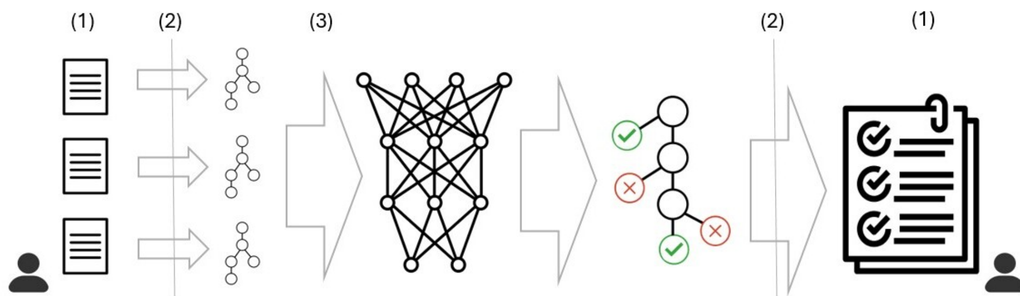

Figure 1: The Pickles testing pipeline: Pickles scenarios (1) are translated (2) into STS, which are then composed (3) into a master model. Formal test cases are derived from the master model, and translated back (2) to Pickles test cases (1).

2.1 Pickles Specification Suite and translation to Symbolic Transition Systems

Listing 1 shows an example of a specification in PicklesDSL. It models a simplified satellite image acquisition system; the spacecraft acquires images over selected Regions of Interest (ROIs), stores the acquired data in on-board memory, and downloads it during contacts with a ground station.

Listing 1: Pickles Specification Suite example.

```
Variable Settings
"memory" is a variable of type decimal with range (0.0, 10000000.0)
"max memory" is a variable of type decimal with range {10000.0}
"active ROI" is a variable of type string with range {Global, South America, Europe}
"image size" is a variable of type decimal with range (0.0, 1000.0)

Scenario 01: Image acquisition in South America
Given the system is in its initial state
And the satellite has an on-board "memory" lower than "max memory"
And the satellite is in an "active ROI" equal to Global
When the satellite enters an "active ROI" equal to South America
Then the satellite should acquire an image with "image size" between 250.0 and 500.0
And the satellite has an on-board "memory" such that:
    "memory" has a value equal to "image size" + stored "memory" AND
    "memory" has a value lower than "max memory"

Scenario 02: No acquisition in Global ROI
Given the satellite is in an "active ROI" not equal to Global
When the satellite enters an "active ROI" equal to Global
Then the satellite has an on-board "memory" such that:
    "memory" has a value equal to stored "memory" AND
    "memory" has a value lower than "max memory"

Scenario 03: Ground station data download
Given the satellite has an on-board "memory" greater than 0.0
When the satellite starts a scheduled contact with a Ground Station
Then the satellite downloads all science products
And the satellite has an on-board "memory" equal to 0.0
```

Scenario 1 models image acquisition when the satellite transitions from the Global ROI to South America and sufficient on-board storage is available: an image of bounded size is acquired and the used memory is increased accordingly, while remaining below the maximum capacity. Scenario 2 captures the complementary case in which, upon entering the Global ROI from any other region, no image is acquired and the on-board memory remains unchanged. Scenario 3 describes data download during a scheduled contact: if stored data are present, all products are downloaded and the on-board memory is emptied. These behaviours are parametrized by the variables in the Variable Settings block. The variables `"memory"` and `"max memory"` represent used and maximum on-board storage, `"active ROI"` denotes the region currently overflown by the satellite, and `"image size"` captures the storage required by an acquired image. Their declared ranges constrain all conditions appearing in the scenarios.

As depicted in our example, a specification suite in Pickles 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. The `Given` clause specifies the precondition of the scenario, that is, the expected state of the System Under Test (SUT) before any interaction occurs. The `When` clause denotes an interaction initiated by an external actor. The `Then` clause describes an observable response of the SUT.

Scenarios may be parametrized using `variables` and guarded by conditions over them. All variables used across scenarios must be declared in the Variable Settings, where their types (e.g. `decimal`) and ranges (e.g. `(0.0, 1000.0)`) 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`.

We express the semantics of our DSL in terms of Symbolic Transition Systems (STS). This formalism extends Labelled Transition Systems (LTS) with data constructs. In STS, it is possible to have variables, both global, called *location variables*, and local to transitions, called *parameters*. Transitions may be augmented with *conditions* over parameters and location variables, as well as with *assignments*, that is, updates to the values of location variables.

In the Pickles framework, each scenario is automatically translated to a STS. The *Given* clause is interpreted as an additional condition on the first transition of the STS. The *When* and *Then* clauses define transitions of the STS, corresponding to input and output actions, respectively. Conditions attached to these steps are translated into transition conditions. All variables in transitions are subject to the type and domain constraints specified in the Variable Settings block. Finally, some scenarios may include the precondition `the system is in its initial state`. This denotes that the scenario can be executed as soon as the system starts; if the condition is not present, we assume that this is not possible.

2.2 Model composition

By applying the translation of Section 2.1 on a specification suite in Pickles, we obtain a set of STSs representing partial specifications of the system's behaviour. We are interested in (automatically) composing such partial models, so that we have a unified model of the complete system behaviour.

A simplified version of the master model resulting from the composition of all three scenarios presented in Listing 1 is introduced in Figure 2. In it, transitions in red correspond to the steps of Scenario 1, those in blue to Scenario 2 and in green are transitions of Scenario 3. We achieve this master model with a combination of two composition operators: choice and sequential composition. Any scenario marked as initial can be chosen as the first to be executed; from then onwards, the sequence may continue with any other scenario. However, tests will only be derived from paths where it is possible to meet all the constraints. For example, the satellite may enter the Global ROI (Scenario 2) after Scenario 1, as the latter implies that the final position of the satellite is the South American ROI (i.e. outside the Global ROI). However, Scenario 2 could not be executed twice sequentially, as the condition stated in its own *Given* would not be met.

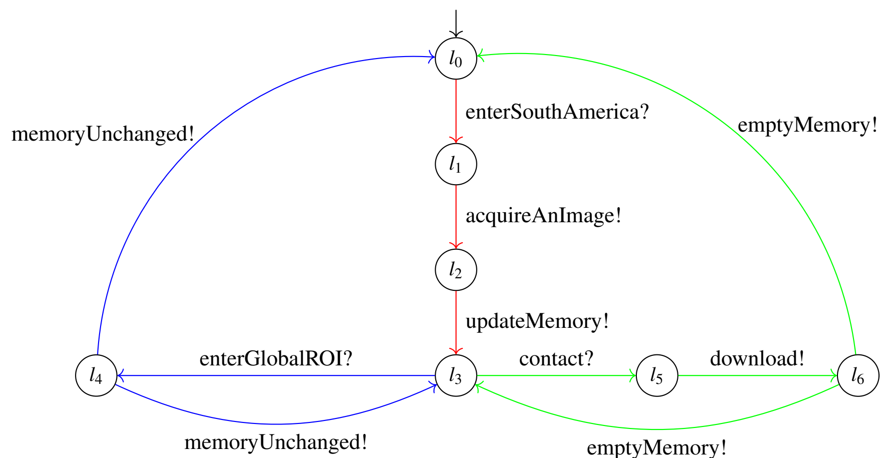

Figure 2: Master model of the Pickles Specification of Listing 1. Transitions in red correspond to the steps of Scenario 1, blue to Scenario 2 and green to Scenario 3. Conditions and assignments are omitted.

2.3 Formal test cases translation to PicklesDSL

Once a full representation of the SUT is obtained as a Symbolic Transition System, we derive concrete test cases using model-based test generation techniques, e.g. for achieving 100% model coverage [6]. We then aim to formally define a mapping that expresses each resulting test case back in Pickles syntax.

Listing 2 presents an example of a test case derived from the master model shown in Figure 2. The test case first assigns initial values to all variables, such that they satisfy the condition of the *Given* step of the first scenario, and then executes a sequence of actions. The test sequentially executes Scenario 1, Scenario 2, and finally Scenario 3. Such a composition is possible because the *Then* step of each scenario satisfies the *Given* step of the subsequent one; we therefore only include the *Given* step of the first scenario and omit the others. For instance, after Scenario 1 the active ROI is South America, which enables the execution of Scenario 2. Similarly, after Scenario 2 the on-board memory is non-zero, enabling Scenario 3, which downloads the products during a scheduled contact. The test case could then continue, for example, with Scenario 1 again, as its *Given* step is satisfied.

The *When* steps in the test differ from those in the specification because concrete values are assigned to the variables, subject to their defined ranges and step conditions. The *Then* steps, however, remain unchanged, as the goal is to evaluate the outputs produced by the SUT; accordingly, the original conditions are preserved. The terminology used in the *When* and *Then* steps (shown in `black text`) is also maintained from the specification (presented in Listing 1) to the derived test cases.

Listing 2: Pickles Test Case Example.

```
Test 01
Given the system is initialized with values:
   "memory" has a value equal to 100.0
   "active ROI" has a value equal to Global
   "max memory" has a value equal to 10000.0
   "image size" has a value equal to 300.0
When the satellite enters an "active ROI" with value South America
Then the satellite acquires an image with "image size" between 250.0 and 500.0
And the satellite has an on-board "memory" such that:
   "memory" has a value equal to "image size" + 100.0 AND
   "memory" has a value lower than 10000.0
When the satellite enters an "active ROI" with value equal to Global
Then the satellite has an on-board "memory" such that:
   "memory" has a value equal to "image size" + 100.0 AND
   "memory" has a value lower than 10000.0
When the satellite starts a scheduled contact with a Ground Station
Then the satellite downloads all science products
And the satellite has an on-board "memory" equal to 0.0
```

3 Related Work

Existing approaches have explored translating natural language (NL) requirements into formal models. For instance, [20] introduces an intermediate language to convert BDD scenarios into formalisms called BDD Transition Systems (BDDTS), but this requires manual translation and additional domain-specific input. Building on this, [19] defines sequential composition for BDDTSs, an idea we adopt for model composition. Other work, such as [8] and [18], automates test generation from parametrized restricted NL specifications, although the resulting test cases are machine-readable only. Structured NL has also been used to formally define requirements. Frameworks such as TEARS [9] and FRET [11] allow formal specification of parametrized timed requirements, yet do not support automated test generation.

Tools like ComMA [15] and TorXakis [16] implement automated test generation with MBT techniques, supporting parametrized and compositional specifications. However, neither the specifications nor the tests are human-readable. Other approaches attempt to produce readable test cases from UML diagrams, for example by extracting tests [5] or generating Gherkin-style criteria [4] from them, but these lack parametrization and are restricted to UML input.

In summary, no existing approach combines parametrized requirement specification, formal semantics, automated test generation and human-readable artifacts, precisely the gap Pickles addresses.

4 Discussion and Future Work

Discussion. The Pickles approach provides several complementary benefits, particularly over traditional BDD-style specification and testing. 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, Pickles specifications are self-contained, condensing 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.

Another advantage lies in the automatic generation of test cases, including input values. This not only provides greater coverage than BDD, but 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 [7]. In addition, automatic test generation also improves maintainability as specifications evolve; test suites can be regenerated automatically, avoiding the need for manual updates and potential inconsistencies. Moreover, test generation heuristics can be adapted to different testing goals, such as generating small suites for smoke testing or large, thorough suites for system testing. As these suites are derived from the same model, they are consistent by construction.

Additionally, explicitly defining variable types and ranges enables early consistency checking at the specification level. Ill-typed expressions or incompatible ranges can be detected before any tests are executed. This feature is particularly useful when different parts of the specification are produced by different teams, as interfaces can be checked early in the development process.

Finally, the automatic composition of scenarios allows the derivation of longer and more diverse test cases that better reflect system usage than isolated scenarios, as typically specified in BDD. This capability 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 [13].

Future Work. In future work we aim to evolve our framework in several directions. To validate Pickles at scale, we will automate the complete pipeline and evaluate its performance across a diverse set of case studies. Additionally, we will conduct user studies to evaluate the developer experience; this process will also allow us to recollect valuable feedback for further improvement.

Moreover, we aim to enrich our DSL with expressions that allow diverse control-flow sequences, including non-deterministic behaviour. We also envision extensions that allow users to annotate scenarios with information about their criticality; such information can be integrated to the test generation algorithm to identify high-risk bugs faster. Given that timing is crucial in critical systems, we also plan to incorporate time-constraint definitions.

Finally, in order to ease the transition from BDD to our framework, we plan to investigate ways to assist this migration. In particular, we will explore how can pre-existing BDD test cases be processed to automatically generalize the expected behaviour of the system from these examples.

Acknowledgements

This research has received support from both the NWO project *Evidence-driven Black-box Checking (EVI)*, with project number OCENW.M.23.155, and the European Union's Horizon 2020 R&D program under the Marie Skłodowska-Curie grant agreement No 101008233 (MISSION project).

References

[1] (2025): *Cucumber*. Available at `https://cucumber.io/`. Accessed: 2025-09-22.

[2] (2025): *Robot Framework*. Available at `https://robotframework.org/`. Accessed: 2025-09-22.

[3] Emil Alégroth, Kristian Karl, Helena Rosshagen, Tomas Helmfridsson & Nils Olsson (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.

[4] Mauricio Alferez, Fabrizio Pastore, Mehrdad Sabetzadeh, Lionel Briand & Jean-Richard Riccardi (2019): *Bridging the Gap between Requirements Modeling and Behavior-Driven Development*. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 239–249, doi:10.1109/MODELS.2019.00008.

[5] Rasha Gh. Alsarraj, Atica M. Altaie & Esra Zuhair Majeed (2025): *Developing an Automated Model-Based Software Testing Tool From the Design Phase*. IEEE Access 13, pp. 58548–58558, doi:10.1109/ACCESS.2025.3553967.

[6] Petra Van den Bos & Jan Tretmans (2019): *Coverage-Based Testing with Symbolic Transition Systems*. In Dirk Beyer & Chantal Keller, editors: Tests and Proofs, Springer, pp. 64–82, doi:10.1007/978-3-030-31157-5_5.

[7] Gul Calikli & Ayse Bener (2010): *Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance*. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE '10, Association for Computing Machinery, New York, NY, USA, doi:10.1145/1868328.1868344.

[8] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta & Mark Blackburn (2014): *NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications*. Science of Computer Programming 95, pp. 275–297, doi:10.1016/j.scico.2014.06.007.

[9] Daniel Flemström, Wasif Afzal & Eduard Paul Enoiu (2022): *Specification of Passive Test Cases Using an Improved T-EARS Language*. In Daniel Mendez, Manuel Wimmer, Dietmar Winkler, Stefan Biffl & Johannes Bergsmann, editors: Software Quality: The Next Big Thing in Software Engineering and Quality, Springer, Cham, pp. 63–83, doi:10.1007/978-3-031-04115-0_5.

[10] Xavier Franch, Cristina Palomares, Carme Quer, Panagiota Chatzipetrou & Tony Gorschek (2023): *The state-of-practice in requirements specification: an extended interview study at 12 companies* 28(3), pp. 377–409. doi:10.1007/s00766-023-00399-7.

[11] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, Julian Rhein, Johann Schumann & Nija Shi (2020): *Formal Requirements Elicitation with FRET*. In: REFSQ Workshops. Available at `https://api.semanticscholar.org/CorpusID:214708107`.

[12] María Belén Rodríguez & Petra van den Bos (2026): *PICKLES: a Natural Language Framework for Requirement Specification and Model-Based Testing*. arXiv:2604.26572.

[13] V Shobha Rani, Dr A Ramesh Babu, K. Deepthi & Vallem Ranadheer Reddy (2023): *Shift-Left Testing in DevOps: A Study of Benefits, Challenges, and Best Practices*. In: 2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS), pp. 1675–1680, doi:10.1109/ICACRS58579.2023.10404436.

[14] Víctor M. Arredondo Reyes, Saúl Domínguez Isidro, Ángel J. Sańchez García & Jorge O. Ocharán Hernández (2024): *Analysis of Behavior-Driven Development: A Thematic Synthesis*. *Programming and Computer Software* 50(8), pp. 701–713, doi:10.1134/S0361768824700713.

[15] Mathijs Schuts, Jozef Hoomann, Ivan Kurtev, Issam Tlili & Erik Oerlemans (2025): *Online Model-Based Testing Reusing Multiple Design Models in an Industrial Setting*. *Journal of Object Technology* 24(2), pp. 2:1–14, doi:10.5381/jot.2025.24.2.a6.

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

[17] Leonardo Villalobos, Christian Quesada López, Alexandra Martinez & Marcelo Jenkins (2019): *Model-based testing areas, tools and challenges: A tertiary study*. *CLEI Electronic Journal* 22, doi:10.19153/cleiej.22.1.3.

[18] Tao Yue, Shaukat Ali & Man Zhang (2015): *RTCM: a natural language based, automated, and practical test case generation framework*. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, Association for Computing Machinery, p. 397–408, doi:10.1145/2771783.2771799.

[19] Tannaz Zameni, Petra Van den Bos, Johan Foederer & Arend Rensink (2025): *Sequential Composition of BDD Transition Systems for Model-Based Testing*. In: *Formal Techniques for Distributed Objects, Components, and Systems*, Springer, pp. 36–54, doi:10.1007/978-3-031-95497-9_3.

[20] Tannaz Zameni, Petra Van den Bos, Arend Rensink & Jan Tretmans (2024): *An Intermediate Language to Integrate Behavior-Driven Development Scenarios and Model-Based Testing*. In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pp. 199–206, doi:10.1109/SANER-C62648.2024.00033.