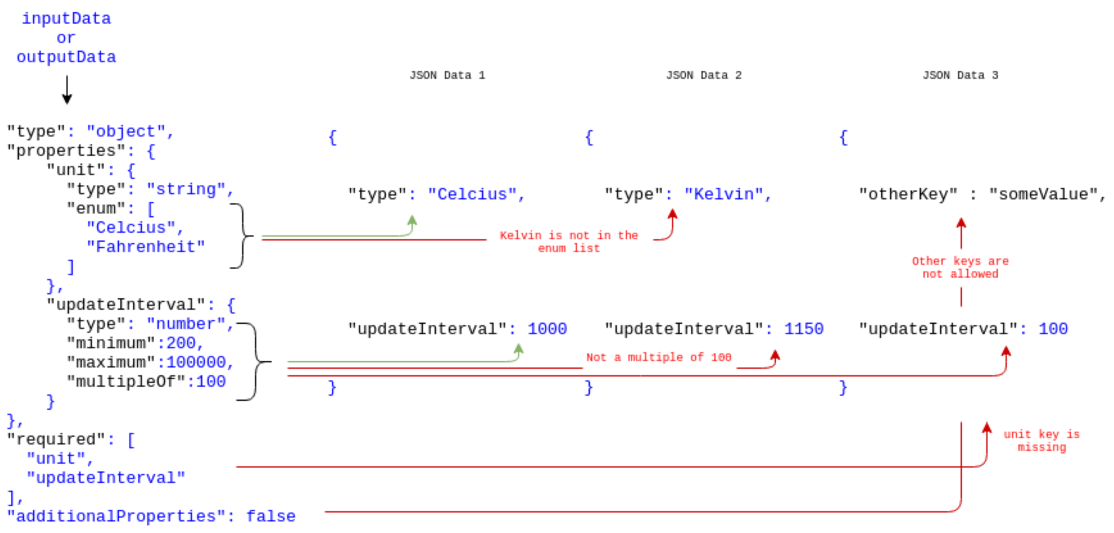
4 Thing Description Based Testing

This chapter will present a novel black-box testing approach using Thing Descriptions. This approach is designed for testing an entire Thing and not its software's intrinsic functions, i.e. no unit testing. In a typical setup, we are expecting to see a physical Thing or its simulation and its TD. Firstly we will discuss the features of a TD that can be used for testing a Thing. Then we will present a test bench architecture that uses the aforementioned features.

4.1 Basics

As mentioned in section 2.1, the interaction patterns of a Thing relies on JSON Schema. This means that once an interaction is executed, it can be only executed with a certain type of data. The same applies to the data returned from an interaction. The TD based testing relies on this principle. This basic validation as defined in [23] is illustrated with an example in Figure 4.1.



```
"type": "object",
"properties": {
    "unit": {
      "type": "string",
      "enum": [
        "Celcius",
        "Fahrenheit"
      ]
    },
    "updateInterval": {
      "type": "number",
      "minimum":200,
      "maximum":100000,
      "multipleOf":100
    }
},
"required": [
  "unit",
  "updateInterval"
],
"additionalProperties": false
```

```
{
  "type": "Celcius",
  "updateInterval": 1000
}
```

```
{
  "type": "Kelvin",
  "updateInterval": 1150
}
```

```
{
  "otherKey" : "someValue",
  "updateInterval": 100
}
```

Figure 4.1: JSON Schema Based Validation Example

In other words, we will test all the possible interactions by manipulating the input data sent and validating the corresponding output data with the JSON Schema defined in the interaction. Executing a series of interactions with different inputs is called a **test scenario**. The aim is to test the Thing with as many test scenarios possible and analyze the test results in order to conclude on how well the specifications, so the TD, are satisfied. This is considered as the core and it is illustrated in Figure 4.2

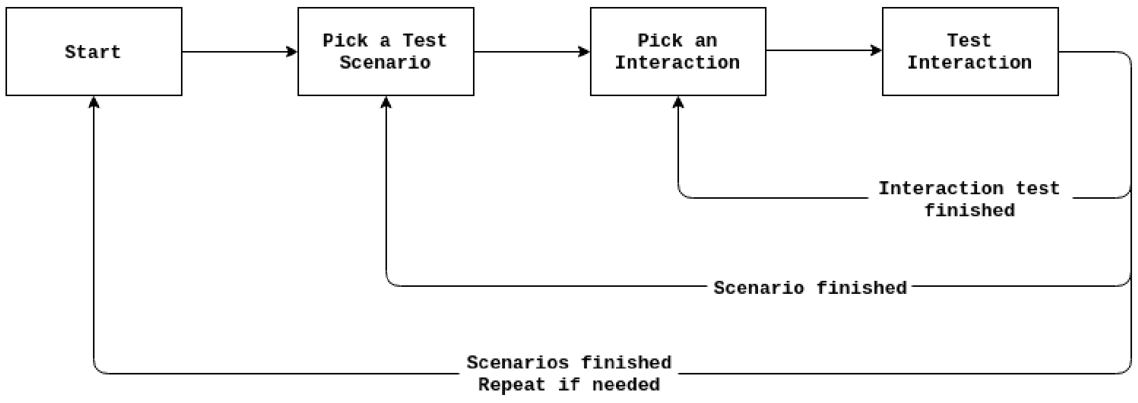


Figure 4.2: Thing Description Based Test Flow Representing the Core

The Section 4.2 details how this testing principle is applied in the context of a complete test bench.

4.2 Architecture of the Test Bench

The validation with JSON Schema shown in Figure 4.1 constitutes the core of the Thing Description based testing; however, there are numerous procedures that need to exist before we can call it a test bench. These include designing a test scenario, analyzing the results and configuring parameters. These procedures are grouped under an architecture and is initially explained with the following steps:

1. Initialize the test bench: The test bench needs to be configured and then initialized. The configuration is a JSON file that specifies the number of repetitions to make, which Thing to test (Thing under Test (TuT)), desired number of scenarios, where to store the results or how to call the test bench. Once the configuration is set, the test bench can be initialized. Upon initialization the Thing under Test is *consumed*. Consuming a Thing is a definition of W3C that means:

   > By consuming a TD, a client Thing creates a runtime resource model that allows accessing the properties, Actions and Events exposed by the server Thing.[13]

   Upon a successful connection to the Thing, the initialization is considered complete.

2. Get the TD of the Thing under Test: Even though consuming Thing includes getting the TD of the Thing, this is still be considered as a different step. In the case of the test bench, TD of the TuT structures the whole test bench. It will constitute the base for generating requests and analyzing the results. At this stage we can also generate the JSON Schemas for each interaction's input and output data if they exist.

3. Generate requests: This is the most *tricky* part of the whole process. Generating requests also defines the test scenarios to be executed. The different methods to generate requests will be explained in Subsection 4.2.1 in order to give enough detail.

4. Test the Thing with each test scenario: This is the core of the testing procedure as it is illustrated in Figure 4.2. Depending on the interactions type, the testing procedure of the interaction differs. For example, if the property is writable then this writability should be tested. All the different methods will be explained in Section 4.2.2.

5. Repeat previous step certain amount of times: Depending on the Thing, testing multiple times with the same test scenario may result in different behavior. It may be because of a test scenario creating a problem that can be found by another test scenario or like in stress testing, a big amount of repetitions can make the Thing fail. Because of these reasons it is possible to choose the number of times to test the same scenarios with the test configuration file.

6. Store and analyze the results: After all the tests are completed, the test bench presents us three ways to interpret the results. Firstly, we can see a table that lists the number of errors occurred in each scenario and its repetitions. This is mainly a quick way to have general view of the results. If there are errors, we need the details on each of them. Each error is presented in the raw test results file which contains all the request-response pairs and the nature of errors. This may be a very big file and may not be easily analyzed by the tester. For this, an analyzed version of the report is also provided. How each report looks like and how the analyzed results look like are explained in Section 4.2.3.

These 6 steps of Thing Description based testing is illustrated in Figure 4.3. Numbers with the same number mean that they are done at the same time and repetitions (step 5) cannot be shown on this diagram but it is simply doing steps 1 to 4 over again.

Following sections detail how requests are generated, how each interaction type is tested differently and how to interpret the results. Chapter 8 provides a testing example that used the technique mentioned in this section.

4.2.1 Request Generation

If the interaction being tested is a writable property or an action that has input data, a certain data has to be transmitted to the TuT. Depending on the scenario and the interaction, the data that needs to be sent will be different. In most testing techniques this has to be done manually. In the case of TD based testing, this can be avoided to some degree.

Listing 4.1 shows how a request data will look like. This type of data may be written manually for each interaction and test scenario. However, Thing Description has the JSON Schema associated to each interaction. This combined with the desired amount of test scenarios in the test configuration file, a template for each interaction of each test scenario can be created. The order of the interactions are created as they appear in the Thing Description but the order can be changed by the tester. The template will contain every key that is

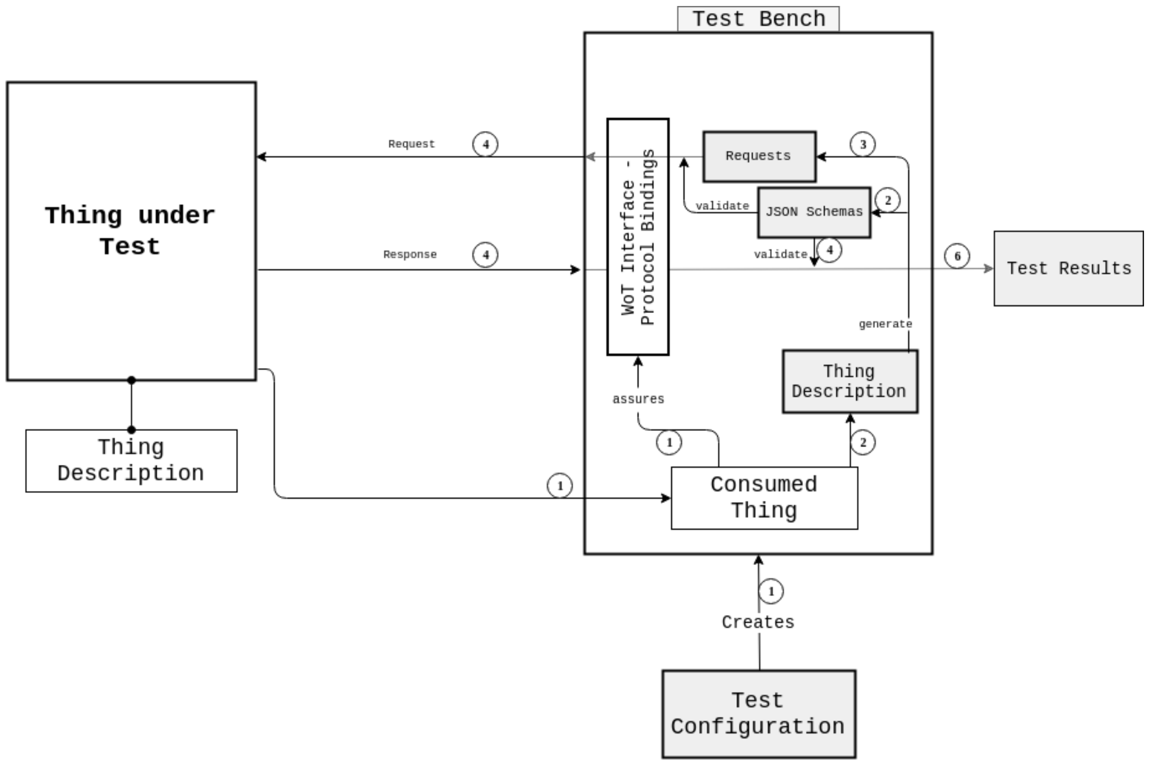


Figure 4.3: Architecture of the Thing Description based Test Bench

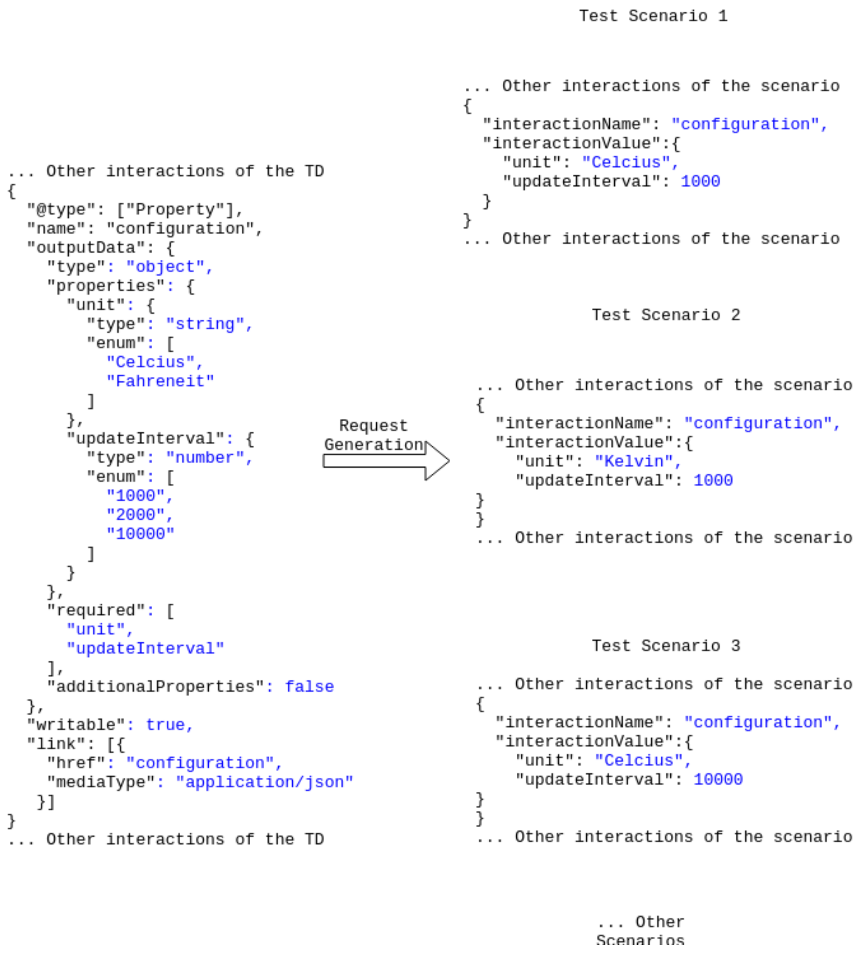
specified in the interaction's inputData, so in Listing 4.1 the text `property1` of `interaction1` will be created but the value `myString10` needs to be written by the tester.

```
{
  {
    "interactionName": "interaction0",
    "interactionValue": "myString00"
  },
  {
    "interactionName": "interaction1",
    "interactionValue": {
      "property1": "myString10",
    "property2": 5
    }
  },
  {
    "interactionName": "interaction2",
    "interactionValue": ["myString30","myString31",4,false    ]
  }
}
```

Listing 4.1: Request Data Example

Another type of request generation is one that is fully automatized. The reason that the previous method still exists is that the fully automatic request generation works only for some cases. In JSON Schema it is possible to give more details on the data to be sent. So in addition to specifying only the type of the data as “string”, “number”, “object” etc., one can also specify fields like maximum value or restricting the values to a set of items. However, these are not mandatory to specify which reduces our request generation capability. Thus only in the ideal case of a fully-specified Thing Description, fully-automated request generation is possible.

For an interaction which has the possible values specified by the `enum` field, we can simply put each of these values in different test scenarios, like in Figure 4.4. For two interactions that have 2 and 3 enum values, we can generate $2 * 3 = 6$ different test scenarios for this interaction by combining each enum value with another. In Figure 4.4 only 3 of them are shown.



```
... Other interactions of the TD
{
  "@type": ["Property"],
  "name": "configuration",
  "outputData": {
    "type": "object",
    "properties": {
      "unit": {
        "type": "string",
        "enum": [
          "Celcius",
          "Fahreneit"
        ]
      },
      "updateInterval": {
        "type": "number",
        "enum": [
          "1000",
          "2000",
          "10000"
        ]
      }
    },
    "required": [
      "unit",
      "updateInterval"
    ],
    "additionalProperties": false
  },
  "writable": true,
  "link": [{
    "href": "configuration",
    "mediaType": "application/json"
   }]
}
... Other interactions of the TD
```

```
... Other interactions of the scenario
{
  "interactionName": "configuration",
  "interactionValue":{
    "unit": "Celcius",
    "updateInterval": 1000
  }
}
... Other interactions of the scenario
```

```
... Other interactions of the scenario
{
  "interactionName": "configuration",
  "interactionValue":{
    "unit": "Kelvin",
    "updateInterval": 1000
}
}
... Other interactions of the scenario
```

```
... Other interactions of the scenario
{
  "interactionName": "configuration",
  "interactionValue":{
    "unit": "Celcius",
    "updateInterval": 10000
}
}
... Other interactions of the scenario
```

Figure 4.4: Request Generation from one Interaction

The other keywords that help us to generate requests are however more generic. These include specifying the minimum and maximum length of a string, minimum and maximum value of a number, regular expressions etc. For generating these requests, a JSON faker is used. The JSON faker tool [24] we use generates random JSON data from a JSON Schema by following all the keywords we have mentioned above. JSON faker is not used for enum type of specification, because if we want to cover all possible combinations of enum values from different interactions, the randomness of JSON faker doesn't guarantee us such coverage. Brute forcing to generate a lot of data that hopefully covers every possible enum value is not an optimal solution either.

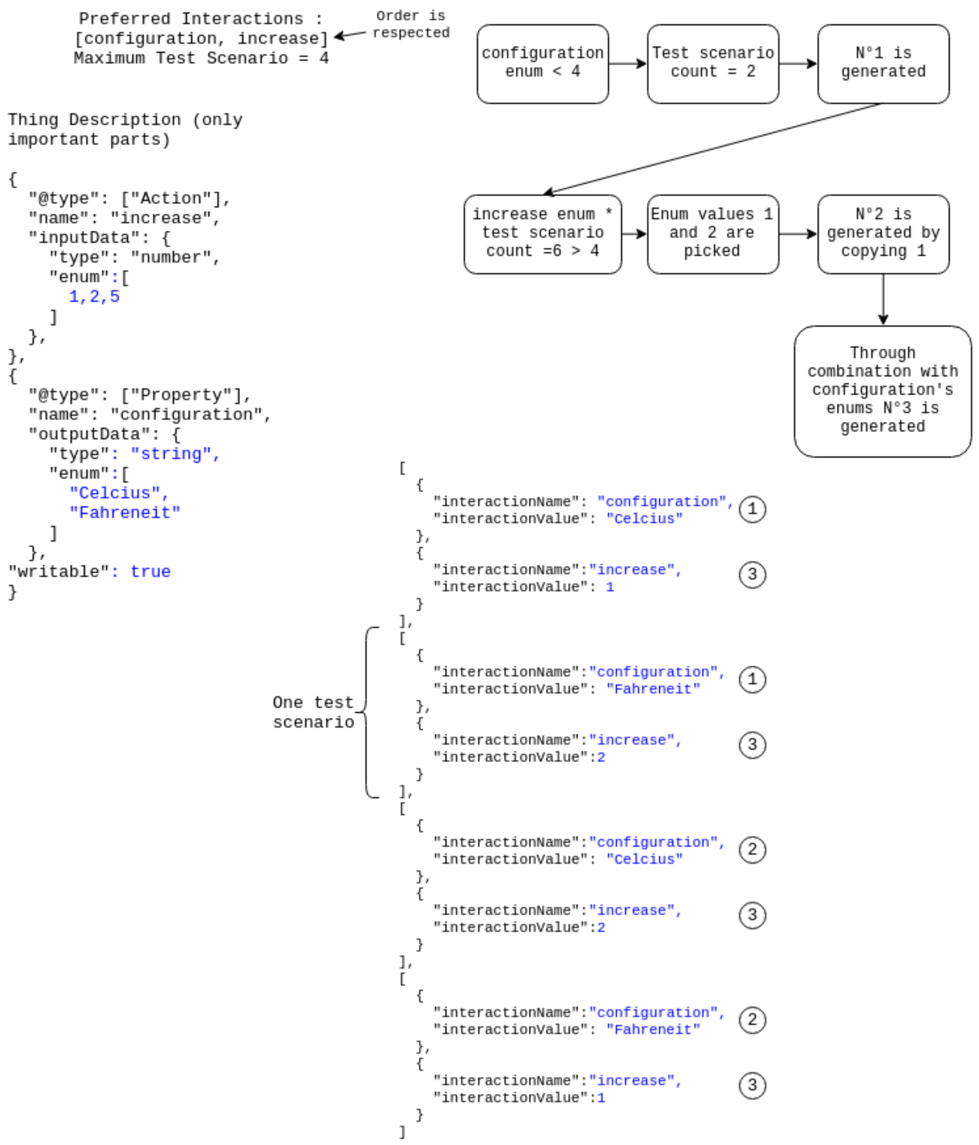
In addition to avoid generating this amount of requests, we have to also avoid generating more test scenarios than the number specified in the test configuration file. This number exists to give the testers the possibility to do quick tests by limiting the number of possible test cases. For this reason, a way to automatically pick combinations that will be generated is proposed. In this generation method it is possible to specify interactions that the test should focus on which we call preferred interactions. By doing so, there will be more combinations of this interaction's data compared to the other ones.

We have grouped most of the generation cases according to 3 conditions:

1. If the number of scenario is one, JSON faker can be used as it is since it is very good for generating one random JSON value. If the number of repetitions specified in the test configuration file is high, this may be compared to using brute force testing to cover all the test cases. This would also achieve 100% automation since the requests doesn't need to be written at all. This is useful for quick testing but cannot be used to cover the test cases in a smart way. This is however only a concept that we haven't implemented because the request generation doesn't happen dynamically at the current version of the test bench.

2. If the test scenario limit is smaller than the possible combinations of the enum values, the order the preferred requests are listed is important. The most preferred request's possible values are compared to the test scenario limit, if it is smaller, all of the enum values are taken as the current test scenario count and we can pass to the next interaction. This next interaction's possible values' size is multiplied by the count value, if it is smaller than the test scenario size we can pass to the next interaction. This goes on until the test scenario limit is about to be surpassed. The interaction that is going to surpass the limit needs to be trimmed, i.e. some enum values are removed for generation. This logic is illustrated in the example of Figure 4.5. During this generation, the non-preferred requests are going to have the same value for all test scenarios.

3. If there are not preferred requests, all the data is generated randomly for each test scenario. The use case for this method would be to generate the wanted amount of requests which later on can be modified by the tester. The unmodified test cases will be random data that the tester is not interested in. The difference of this case from the first one is that in the first case we have no control over the data being sent. Differently, in this case, some of the randomness can be corrected by the tester.


Preferred Interactions :
[configuration, increase]
Maximum Test Scenario = 4
Order is respected
configuration enum < 4
Test scenario count = 2
N°1 is generated
increase enum * test scenario count =6 > 4
Enum values 1 and 2 are picked
N°2 is generated by copying 1
Through combination with configuration's enums N°3 is generated
Thing Description (only important parts)
One test scenario
1
3
1
3
2
3
2
3

```
{
  "@type": ["Action"],
  "name": "increase",
  "inputData": {
    "type": "number",
    "enum":[
      1,2,5
    ]
  },
},
{
  "@type": ["Property"],
  "name": "configuration",
  "outputData": {
    "type": "string",
    "enum":[
      "Celcius",
      "Fahreneit"
    ]
  },
"writable": true
}
```

```
[
  {
    "interactionName": "configuration",
    "interactionValue": "Celcius"
  },
  {
    "interactionName":"increase",
    "interactionValue": 1
  }
],
[
  {
    "interactionName":"configuration",
    "interactionValue": "Fahreneit"
  },
  {
    "interactionName":"increase",
    "interactionValue":2
  }
],
[
  {
    "interactionName":"configuration",
    "interactionValue": "Celcius"
  },
  {
    "interactionName":"increase",
    "interactionValue":2
  }
],
[
  {
    "interactionName":"configuration",
    "interactionValue": "Fahreneit"
  },
  {
    "interactionName":"increase",
    "interactionValue":1
  }
]
```

Figure 4.5: Request Generation Logic with Preferred Interactions

Finally, all the different method is summarized in Table 4.1.

Table 4.1: Comparision of Different Generation Methods

Generation Method	**Advantages**	**Disadvantages**
Manual	• Complete control over the generation • Good for copy pasting from already existing scenarios	• Labor intensive • Error prone
Template from Thing Description	• Good for adapting the test scenario that is written in another document without making format errors	• Labor intensive
Automatic with Test Scenario =1	• Completely automatized • Good for stress testing	• Excessively random • Cannot edit the requests being sent
Automatic with Preferred Interactions	• Focusing the test on a group of interactions • Automatic	• Random data exists partially • Small input required to plan the preferred interactions
Automatic without Preferred Interactions	• A template with existing values • Automatic	• Excessively random if not edited

4.2.2 Different Types of Interaction Testing

The Thing Description lists different interaction types that needs to be tested differently. This study focuses only on *Property* and *Action* types which result in 6 different testing procedures:

1. Property, not writable: In this case, the property value is requested and it is compared to the JSON Schema. Only JSON Schema validation shows that this test has not passed.

2. Property, writable: In addition to the previous case, after getting the property value, a write operation is performed. The write value comes from the generated requests and is validated against a JSON Schema before sending it. After writing, the same property is read, *hoping* to get the same value back. If the value changes between two requests, it is considered as an error but is written to the results file with a different error id. However if the property value is still the same as the value gotten at the first get, it is considered as a true error.

3. Action, with no input and no output data: This is the most simple action testing. The action is tested only to see if it exists and doesn't return anything. If it returns something, it is written as a separate error case, i.e. not a JSON Schema validation error.

4. Action with no input data but with output data: In addition to the previous one, the response is validated with the JSON Schema.

5. Action with input data but no output data: The input data is taken from the generated requests and is validated against its JSON Schema. This validation exists since the requests can be manually written by the tester. Then the validated request is sent to the Thing to invoke the action.

6. Action with both input and output data: In addition to the previous one, the response is also validated against a JSON Schema.

The property and action testing are illustrated by Figures 4.6 and 4.7 respectively. All of the rectangles (nodes) can result in an error. This can be because of the Thing not responding to the request, request not being generated or most probably the response that is not valid against its JSON Schema. In case of an error, the interaction test is finished by recording the error, like shown in Listing 4.2

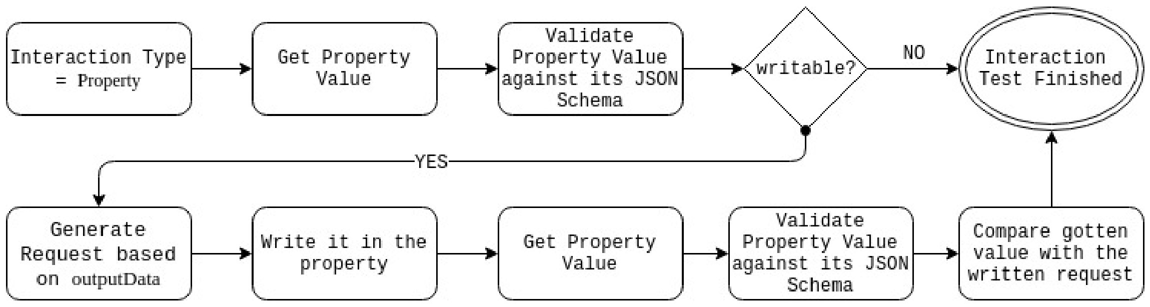

Interaction Type = Property
Get Property Value
Validate Property Value against its JSON Schema
writable?
NO
Interaction Test Finished
YES
Generate Request based on outputData
Write it in the property
Get Property Value
Validate Property Value against its JSON Schema
Compare gotten value with the written request

Figure 4.6: Flowchart for Testing Property Interactions

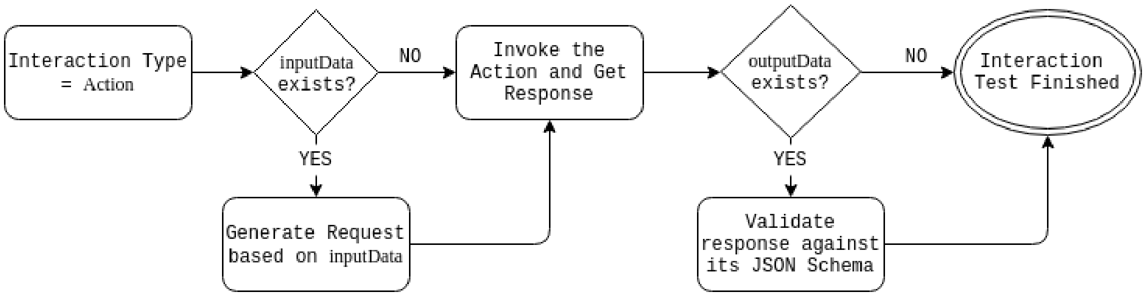

Interaction Type = Action
inputData exists?
NO
Invoke the Action and Get Response
outputData exists?
NO
Interaction Test Finished
YES
Generate Request based on inputData
YES
Validate response against its JSON Schema

Figure 4.7: Flowchart for Testing Action Interactions

4.2.3 Results of a Test

During the testing, each request response pair is recorded with the format shown in Listing 4.2.

```
{
  "name": "increase",
  "result": false,
  "sent": 5,
  "received": 38,
  "errorId": 16,
  "error": " Received response is not valid: Expected boolean
     return value

}
```

Listing 4.2: Result of One Interaction Test

These values are put in a file that can be read by the tester; however, it may be time consuming once the are many test scenarios and repetitions with different outcomes. One simple way to provide some information is grouping the results in a table, showing the failure for each. Table 4.2 illustrates such a table.

Table 4.2: Simple Table Summarizing the Test Results

Test Scenario Number	**TS1**	**TS2**	**TS3**
Repetition Nb			
R0	0/3	2/3	1/3
R1	0/3	2/3	1/3
R2	0/3	2/3	1/3
R3	0/3	2/3	1/3
R4	0/3	2/3	1/3

In Table 4.2 we see that we have to don't have to investigate the first test scenario and that we should focus on the last one to see what is the problem. So this table cannot be used to asses the individual problems in each request-response pair. After seeing this table, the tester can look at the raw test results file to understand the details. However, this can still be a time consuming task. We can facilitate this by analyzing the test results using the Thing Description.

During the implementation of the test bench, we have used it against Things of different coding quality and progress. This has allowed us to analyze the common problems. The most frequent test the Thing cannot pass is the return value that doesn't validate the JSON Schema. Also in some cases the written property is not the same after the second time it is read. Generally the repetitions don't provide very different results, unless the requests are sent truly randomly. The following list explains what can be processed from the raw test results file in order to give important insights:

- List of interactions that cannot be executed because of not accessing them.
- Whether the repetitions give the same results. The test scenarios that give different results are listed with their different result-response pairs, similar to the raw results file.
- If there is a sudden increase in errors after a repetition.
- If an interaction always or mostly produces an error.
- If a property always or mostly doesn't give back the same value written.
- If a writable property doesn't accept a write operation.
- If a particular test scenario has nearly only errors.
- If an interaction has passed all the tests

The true goal of the analyzed version of the test results, is to better plan the upcoming tests. If the results show that an interaction fails most of the tests, in the next test the tester can focus on this particular interaction by preferring this interaction in the generation process (see Section 4.2.1) and removing an interaction that passes all the tests from the preference list. The use of analyzed results file will be demonstrated in Chapter 8.

4.2.4 Test Bench as a Thing

Until now, the test bench concept is presented as a usual software. However in this section we will represent how the test bench can be a Thing itself and why it would be interesting.

In Section 4.2.2 it is said that the property's value can change between the write and the second read request. This can be due to a bug in the software; however, it is most probably because of an event occurring between the two requests, like the temperature increasing just enough to pass to the next integer value. This can be solved by locating the test bench *close* to the Thing under Test, for example as another Thing of the system .

In addition, modeling the test bench with a TD would give us the same advantages that are brought by using TDs in development process like explained in Section 3.2. This is achievable since the test bench is also a Web client whether it is a Thing or not. But if the TD is used for its development, the effort to make it a Thing is minimal. Furthermore, this means that the test bench would be a Web service that anyone may develop a front-end Web application for.

The Listing 4.3 shows the TD of the test bench. It is possible to :

- Set the configuration of the test bench
- Initialize the test bench
- Generate the requests if it is fully automated or provide the requests to be sent
- Test the Thing
- Get the test results

```
{
  "@context": ["http://iot.linkeddata.es/def/wot#"],
  "@type": ["Test Bench"],
  "name": "thing_test_bench",
  "base": "http://localhost:8090/",
  "interaction": [
    {
      "@type": ["Property"],
      "name": "testConfig",
      "outputData": {
        "type": "object",
        "properties": {
          "thingTdLocation": {"type": "string"},
          "thingTdName": {"type": "string"},
          "schemaLocation": {"type": "string"},
          "testReportsLocation": {"type": "string"},
          "requestsLocation": {"type": "string"},
          "repetitions": {"type": "integer","minimum": 1},
          "maximumTestScenarios": {"type": "integer","minimum": 1}
        },
        "additionalProperties": false
      },
      "writable": true,
      "link": [{
          "href": "testConfig", "mediaType": "application/json"
      }]
    },
    {
      "@type": ["Action"],
      "name": "initiate",
      "outputData": {"type": "boolean"},
      "link": [{
          "href": "initiate", "mediaType": "application/json"
      }]
    },
    {
      "@type": ["Action"],
      "name": "generateRequests",
      "inputData": {
        "type": "string",
        "enum": ["manual","template","automatic"]
      },
      "link": [{
          "href": "generateRequests", "mediaType": "application/json"
      }]
    },
    {
      "@type": ["Property"],
      "name": "requests",
      "outputData": {"type": "array"},
      "writable": true,
      "link": [{
          "href": "requests", "mediaType": "application/json"
      }]
    }
    {
      "@type": ["Action"],
      "name": "testThing",
      "inputData": {"type": "boolean"},
      "outputData": {"type": "boolean"},
      "link": [{
          "href": "testThing", "mediaType": "application/json"
      }]
    },
    {
      "@type": ["Property"],
      "name": "testReport",
      "outputData": {"type": "array"},
      "writable": false,
      "link": [{
          "href": "testReport", "mediaType": "application/json"
      }]
    }
  ]
}
```

Listing 4.3: Thing Description of the Test Bench

Another interesting point about the future use of test bench disguised as a Thing will be also given in Section 10.2.

4.3 Summary of This Chapter

In this chapter we have introduced a novel architecture to use Thing Descriptions to build a test bench. With this architecture we are able to test actions and properties of a Thing in a black-box testing approach. The architecture was based on JSON Schema but generating requests was a more difficult task. We have proposed different methods and compared them so that a tester can choose one accordingly.

Lastly we have shown that this architecture can be a Thing on its own. We have presented the advantages of adopting such an approach, like accessing a test bench from distance with our own front-end application.

In the following chapters we will detail how to automatically create a Thing that can be tested by this approach and how it can be applied to a real life use case.