ISSN 1884-0760

# GRACE TECHNICAL REPORTS

# Enlightening Test-Driven with Formal, Formal with Test-Driven through Spec-Test-Go-Round

Fuyuki Ishikawa, Takuo Doi, Kazunori Sakamoto,
Nobukazu Yoshioka, and Yoshinori Tanabe

GRACE-TR 2015–05 June 2015

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

**WWW page: http://grace-center.jp/**

# Enlightening Test-Driven with Formal, Formal with Test-Driven through Spec-Test-Go-Round

Fuyuki Ishikawa[1], Takuo Doi, Kazunori Sakamoto[1],
Nobukazu Yoshioka[1], and Yoshinori Tanabe[2]

[1] National Institute of Informatics, Japan
f-ishikawa@nii.ac.jp
[2] Tsurumi University

**Abstract.** The impression is that formal methods are at odds with popular approaches such as agile software development. However, these different approaches should instead be significant mutual complementary in their essential principles. It is an essential problem how to convince this point beyond apparently different communities that tend to be disjoint. This paper discusses this point by combining the principles of test-driven development and its extensions (such as "specification by example") with formal specification methods. We constructed a language and prototype tool called JSTN that mixes specifications, test designs, and test cases with a model finder. Together, they enable interactive, quick feedback as well as a flexible blend of different principles. The role of this tool in education and enlightenment is evaluated through a seminar with over 60 industry engineers including many "agile people." [3]

**Keywords:** Software Engineering Education, Formal Specification, Test-Driven Development, Specification by Example, Test Generation, Model Finding, Alloy

## 1 Introduction

There has been a long discussion on how to position formal methods among a variety of approaches in software engineering. Specifically, the recent directions of agile software development (ASD) have prompted various "agile versus formal" debates [6]. However, the objective and obvious conclusion should be that the two paradigms can benefit each other a lot if we focus on their essential principles. The essential problem is how to convince this point beyond apparently different communities that tend to be disjoint.

One of the key principles of formal methods is the use of properties or propositions. The declarative style facilitates a focus on "what" while avoiding unnecessary decisions or descriptions of "how." Rigorous and machine-readable descriptions remove ambiguity and open up various possibilities of tools.

[3] This is a full version of the paper under submission to The 17th International Conference on Formal Engineering Methods.

From the same aspect of specifications, ASD communities have investigated test-driven development (TDD) [5]. Here, test cases (or examples) are considered primary elements that guide the development activities. They play similar roles to those of specifications, and are not only used to check the constructed code. Recent extensions, such as behavior-driven development (BDD), have honed this direction with principles such as "tests as documents" and "specification by example" [2, 4]. Test cases can give more confidence and have less dependency on experience of programming or declarative descriptions.

It is somewhat obvious that properties and test cases help to derive or validate each other. For example, to validate properties, it is necessary to consider both cases that satisfy properties and those that do not. Because of the nondeterminism of properties, it is also necessary to distinguish "one case among acceptable ones" from "the only acceptable case." Another essential aspect is to allow for specifying test designs, or properties on the test suite (the set of test cases), which describe the rationales or intentions behind the test cases.

To investigate this state of affairs, this paper uses a language that includes syntax for all of the formal specification, test designs and test cases. With a model finder tool wrapping Alloy Analyzer [1], it enables a flexible blend of principles from different communities, agile and formal, as well as testing for quality assurance. It also allows for a flexible blend of automated, formal ways and manual, informal ways. The role of this tool in education and enlightenment is evaluated through a seminar with over 60 industry engineers including many "agile people."

The remainder of this paper is organized as follows. Section 2 discusses the principles and difficulties of formal specification and TDD. Section 3 presents the proposed language and tool JSTN. Section 4 reports seminar experiences with industry engineers. Section 5 evaluates the proposal through a discussion of related work, before the concluding remarks in Section 6.

## 2 Preliminary

This section discusses the principles and difficulties with properties and test cases for specification. We use a popular sample for test design, that is, a function that judges type of a triangle given three integer values for the lengths of the edges [15]. Below is the interface definition in the Java language. The result type is defined by an enum type for equilateral, isosceles, scalene, and non-triangle.

```
enum TType{ EQUI, ISO, SCA, NON }
TType judgeTriangle(int a, int b, int c);
```

### 2.1 Specification by Property

We assume that readers will know the various usages and benefits of formal specification, e.g., random testing with enough assertions [20], in static analysis on the code [7]. What follows therefore deals only with its difficulties.

The specification of this sample function can be given in the form of a precondition and postcondition, on the basis of Hoare logic. Suppose an engineer specified the following condition as part of the postcondition.

```
$TRI && a != b && b != c ==> \result == TType.SCA
$TRI && ( ... || c == a && a != b) ==> \result == TType.ISO
```

`$TRI` denotes the abbreviated condition to compose a triangle, the line break means conjunction, the symbol `==>` denotes the logical implication, and the keyword `\result` refers to the return value. There is a deficiency: the second line forgets to mention the condition `c != a`, leading to too strong a condition.

First suppose the engineer uses this postcondition to assert executable description, e.g., as in VDM [8] or JMLUnit [20]. He/she gives a behavior specification or implementation of the function, then designs and runs test cases to check the results satisfy the postcondition (as well as assertions given in each test case). For example, the following test case fails due to the above deficiency.

```
assert( judgeTriangle(5,3,5), TType.ISO )
```

If `c == a && a != b`, the postcondition check always fails because it requires both `\result == TType.SCA` and `\result == TType.ISO`. Although this will reveal the deficiency, it is confusing as the engineer intended to find deficiencies in the behavior specification or implementation, not the postcondition.

In contrast, the engineer can be more careful and test the postcondition function directly:

```
assert(post_judgeTriangle(5, 3, 5, TType.ISO), true)
```

This test case fails and the deficiency is revealed more clearly, because the possible causes are localized (like unit testing before integration testing).

Second, let us the engineer uses a model finder such as Alloy Analyzer [1]. It can present possible models (examples) that satisfy the given condition, in this case for `(a,b,c,\result)`, such as `(3,3,3,Type.EQUI)`, `(4,4,2,Type.ISO)` (within some bounds). However, it does not show any values that satisfy `c == a && a != b` (without any notice). An inconsistency is reported only if the engineer explicitly tells the finder to show values for `c == a && a != b`.

In many cases with complex problems, it is more likely to have a weak postcondition while strong one is more effective [18], for example:

```
\result == TType.SCA ==> a != b && b != c
```

This condition is weak, missing `c != a` as well as the condition to compose a triangle (`$TRI`).Postcondition weaker than the necessary and sufficient condition can be acceptable in some usages, e.g., assertions. However, the above case should be due to oversight, accepting a result `TType.SCA` for `(a,b,c)=(5,3,5),(2,3,7)`.

If the engineer use the weak postcondition to assert executable description, the deficiency may not be revealed. The problem is the postcondition silently lets incorrect results pass, e.g., `TType.SCA` for `(a,b,c)=(5,3,5)` is accepted. If the engineer tests the postcondition directly, test cases will reveal the deficiency. with test cases of values that do not satisfy the postcondition, such as:

```
assert(post_judgeTriangle(5, 3, 5, TType.SCA), false)
```

A model finder requires good fortune to generate unexpected examples, as well as carefulness of the engineer to notice them. The deficiency is revealed as an inconsistency if the engineer explicitly gives and asserts that such counterexamples do not satisfy the condition.

The above discussion illustrates it is very important to validate properties with concrete test cases or examples in a careful test design. Test designs in this context should not only cover different situations (e.g., scalene triangles) but also handle the nondeterministic nature of the properties (i.e., assertions are different from those for program code and may not constrain the result uniquely).

### 2.2 Specification by Example

If the engineer uses test-driven development for the same function, he/she will start with a check list and test cases. The (initial) check list probably includes items like "Judge Equilateral," "Judge Scalene," etc. If the engineer thinks the equilateral item is the easiest, he/she would design a test case for it first:

```
assert( judgeTriangle(5,5,5), TType.EQUI )
```

The engineer sees that this test case fails, and concentrates on writing code to pass it. The code can even be fake; e.g., just write `return TType.EQUI`, if he/she thinks the problem is too difficult to write clean code that works. After passing the first test case, the engineer chooses another item in the check list and writes a test case, e.g., for a scalene triangle, which fails. This small cycle is repeated until the people involved have confidence about realization of the function. The engineer may extend the check list as needed.

The first point is to have quick feedback with small cycles. In other words, one should avoid writing long code without any checks, as it can result in code that mysteriously works for some test cases, but not for others. Such code entails a lot of effort to debug and rollback. The second point is to use and discuss concrete test cases, or examples. Even customers (non-programmers) can understand and discuss the examples with confidence, if it is the UI-level function. Even experts may assign different meanings to general expressions, or easily overlook deficiencies, due to careless mistakes or the complexity of the problem[4].

These points suggest supporting formal specification by TDD. It can be attractive to apply the TDD principles to formal specification, e.g., for relatively new, open problems with some complexity that requires concrete examples. In such case, it is necessary to consider the meaning of "test" to match with natures of properties as discussed in Section 2.1.

On the other hand, TDD completely relies on the design of the test cases. Although a check list can start with easy examples, it should eventually become

[4] We will omit discussing other essential aspects of TDD and its extensions, such as focus on the value without unnecessary generalization or extension, refactoring, or human-readable representation of test cases [2, 4, 9].

complete enough for the customer to have confidence in achievement of the expected value. The fundamental methods for testing are helpful, e.g., equivalence partitioning and boundary analysis, but require context-specific applications and discussions (e.g., what are the partitions in our case?). The test cases such as `(a,b,c)=(5,3,5)` are actually derived results. The test designs, or the intentions behind them, is a target of discussion and validation (e.g., try one case of `c == a && a != b`). Documentation is indispensable especially if future changes are expected. Even with documentation efforts, the understanding and validation of test designs can suffer from the ambiguity of natural languages and complexity of logic. It is a costly and error-prone undertaking to manually trace and validate test cases for the test design.

It seems to be a good idea to support TDD by formal specification or properties. The difficulties discussed above are similar to difficulties of specifications, which are typical motivations for formal specification. Actually, what we call "test design " is a specification of the test suite, or a set of test cases, e.g., "include one test case for isosceles triangles with `a == c`." The specification of the function also matters as a test oracle that determines whether the expected output values are correctly defined. Thus it is valuable to derive or assert test cases with formal properties as needed.

Testing for quality assurance explores more kinds of completeness, such as coverage on combinations even without causal relationships (e.g., pairwise testing) [13]. This paper does not discuss or evaluate such aspects, though the proposed language and tool can potentially handle them.

## 3 JSTN Language and Tool for Spec-Test-Go-Round

The proposed language and tool are collectively called JSTN (Java Specification and Testing Note). The JSTN language is designed for annotations on the Java language, and hence it will be use to a wide range of engineers, though the concepts involved do not essentially depend on Java. The language has vocabularies for the specification (in the form of properties), test designs and test cases. The semantics are defined from the viewpoint of a generator of test cases or examples.

### 3.1 Sample

Here we illustrate the language and tool with the triangle sample in Section 2. We start with specification of a precondition, which was not considered in Section 2. Suppose that non-positive values for the input `(a,b,c)` are considered invalid, rather than valid input that makes a non-triangle. In this case, the tool generates test cases that include only positive values for the three arguments:

```
\pre { a > 0 && b > 0 && c > 0 }
```

| a | b | c |
|---|---|---|
| 2 | 3 | 10 |
| 19 | 20 | 0 [LowerB] |
| ... | ... | ... |

The choices of the values are arbitrary. In the second line, the value for `c` accompanies a tag "LowerB" (for lower boundary). In this simple case, the tool can understand the constraints bound by constant values and can attach the tag. The tool may generate the output value (`TType`) as well, but does not do so by default as arbitrarily wrong output values are not so meaningful.

The tool also accepts a command to generate test cases with invalid input, attaching tags for invalid values:

| a | b | c |
|---|---|---|
| -2 [Under] | 10 | 3 |
| -1 [UnderB] | 5 | -8 [Under] |
| ... | ... | ... |

Suppose we add part of the postcondition for equilateral triangles. The tool then starts to include the result in the output (for the valid input):

```
\post { a == b && b == c ==> \result == TType.EQUI }
```

| a | b | c | \result |
|---|---|---|---|
| 3 | 3 | 3 | TType.EQUI |
| 1 [LowerB] | 3 | 5 | TType.NON |
| 10 | 9 | 3 | TType. EQUI |
| ... | ... | ... | ... |

As expected, result values are arbitrary for all cases except for the first row that matches the left side of the implication (`a == b && b == c`). The tool applies a heuristic that automatically adds a test design that includes at least one case that matches the left side of top-level implication formula.

Suppose the engineer had a test case already defined and agreed with related people (e.g., consumers of this function). He/she can add the test case with a tag and the case is then always included in the result of the tool with the tag:

```
\case{ \"ex-equi" a==5 && b==5 && c==5 && \result==TType.EQUI }
```

| a | b | c | \result | PROP |
|---|---|---|---|---|
| 5 | 5 | 5 | TType.EQUI | [ex-equi] |
| 3 | 3 | 3 | TType.EQUI | |
| 1 [LowerB] | 3 | 5 | TType.NON | |
| 10 | 9 | 3 | TType. EQUI | |
| ... | ... | ... | ... | ... |

Confidence in the meaning of the postcondition can be increased by using a command to show counterexamples, or wrong execution results from valid input:

| a | b | c | \result |
|---|---|---|---|
| 3 | 3 | 3 | TType.ISO |
| 5 | 5 | 5 | TType.NON |
| 2 | 2 | 2 | TType.SCA |
| ... | ... | ... | ... |

We proceed to the next step in the TDD way, adding a test case of scalene triangles, which is not specified yet:

```
\case{ \"ex-sca" a==5 && b==3 && c==4 && \result==TType.SCA }
```

This test case does not cause a failure, as the current postcondition does not allow any result for the input. The tool includes this test case in the result table (omitted) in the same way. It is also possible to strongly assert that this example defines the only possible result from the input by `example_d` (for "deterministic"), which lets the tool reports the inconsistency.

From a different aspect, the language allows to give test design descriptions, primarily in the form of partitions. The following is a sample of a test design description, that is not only partial but also naive.

```
\partition{
  \"p-equi" a == b && b == c, \"p-sca" a != b && b != c,
  \"p-iso1" a == b && b != c, \"p-iso2" b == c && c != a,
  \"p-iso3" c == a && a != b
}
```

The scalene partition lacks `c != a` and thus is not disjoint with the third isosceles partition. The following is a possible result table:

| a | b | c | \result | PROP |
|---|---|---|---|---|
| 3 | 4 | 5 | TType.EQUI | [p-sca] |
| 5 | 5 | 4 | TType.ISO | [p-iso1] |
| 9 | 3 | 3 | TType.NON | [p-iso2] |
| 4 | 7 | 4 | TType.NON | [p-sca,p-iso3] |
| 5 | 5 | 5 | TType.EQUI | [ex-equi,p-equi] |
| ... | ... | ... | ... | ... |

The tags help a lot to see which case is in which partition(s), and in this case there are unexpected partition tags in the fourth row. Actually the above partition also lacks the condition to compose a triangle, which may be found similarly when the partition for non-triangle is added.

Figure 1 illustrates how JSTN allows for partial or incremental construction of specifications, test designs and test cases. At the top is a situation where no specification is given other than the type information. The tool can only enumerate possible values in the type. The bottom left shows a situation where the complete specification of the function is given. In this situation, the tool can exactly judge which cases are valid and which cases are invalid. The bottom right shows a situation where a complete test design is given on the test cases. The tool can, for example, select only one case in each partition and include boundaries if necessary. The middle situation on the left side is one where the specification of the function is partially given. In such situations, the tool would judge some cases that are actually invalid as valid. Similarly, the middle right situation is one where the test design is partial. Here, the tool can only generate necessary test cases or eliminate duplicate cases only for that part.

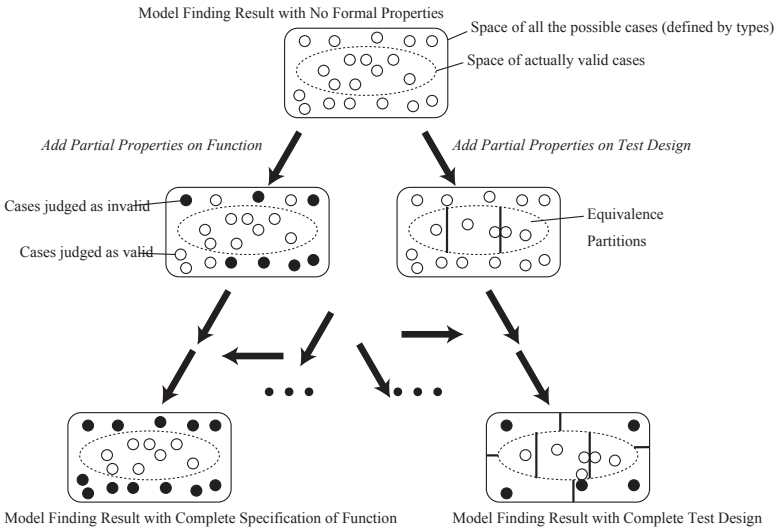


**Fig. 1.** Balance of Formal Properties and Tool Result

The JSTN tool can generate test cases from partial descriptions, allowing to validate what was just specified and have confidence about them. This quick feedback means the tool accepts partial specifications, for the purpose of semi-automation or making weak assertions. It also means the tool allows for an incremental process with continuous validation, toward a complete specification or a complete test design. We call the process "Spec-Test-Go-Round", in which one investigate specifications and test cases (boundary lines and instance circles in Figure 1) in turn. The JSTN tool provides functionalities to go not only from specifications to test cases but also in the reverse direction, e.g., selecting test cases to use hereafter from the generated ones.

### 3.2 Supported Tasks

The sample in 3.1 showed various features of the language and tool, however, lacked objectives or consistent processes. Actually it is necessary to clarify the deliverable under consideration and how to use the language and tool.

JSTN can be used for tasks to construct test cases. One can provide concrete test cases directly, use test designs to generate test cases, or mix them. It is possible to mix test designs and test cases in a complementary way for the purpose of semi-automation. Part of test cases are generated while the others are manually described. For example, it is easier to generate test cases that follow some clear rules, e.g., partitions and boundaries by integer constants. However, it is much more difficult to provide properties that are strong enough to generate

test cases corresponding to specific structures and value distributions of trees. In addition, it is also possible to mix test designs and test cases in a duplicated way for the purpose of validation, i.e., asserting test cases described manually by test designs. This increases confidence in test cases and test designs (as deficiencies can occur in either) and also enables to keep the traceability. Specifications of the target functions can also work as assertions on test cases.

JSTN can be used also for tasks to construct specifications. One can validate a specification by providing concrete test cases or generating them. It is up to the engineers to follow the TDD way or test-first principles. or not. The JSTN language includes syntax specific to assertions for properties, i.e., determinism and counterexamples. In addition, the generated test cases or counterexamples help to increase confidence when they were found to follow the specification as expected. There is also the possibility of finding unexpected cases arising from too strong or too weak specifications.

### 3.3 Core Syntax Elements

Here, we explain the core syntax elements and their semantics for the case the tool is used to show test cases about a method. We will omit explaining those when the tool is used to show examples of variable (field) values that satisfy (or do not satisfy) invariants.

Let $TS$ be a test suite, or a set of test cases to be generated. Each test case $t \in TS$ consists of input and output values $t = \{i_1, \ldots, i_m, o_1, \ldots, o_n\} \in I_1 \times \ldots \times I_m \times O_1 \times \ldots \times O_n$. The domains of the input values $(I_1, \ldots, I_m)$ refer to the types of the method arguments and pre-state variables that appear at least in one of the annotations (e.g., postcondition)[5]. Thus JSTN deals with variables, in or out of the class the target method belongs to, only if they are mentioned in the annotations. Similarly, the domains of the output values $(O_1, \ldots, O_n)$ refer to the types of the return value and post-state variables that appear at least in one of the annotations[6]. The actual domains depend on the implementations that use bounds to avoid state explosions.

There are four modes of generation, *valid*, *invalid*, *checkpost* and *checktest*. The semantics for each type of annotations are explained below.

Suppose that $PRE = \{pre_1, \ldots, pre_i\}$ and $POST = \{post_1, \ldots, post_j\}$ are sets of given precondition predicates and postcondition predicates, respectively.

- The *valid* mode generates $TS$ such that $\forall t \in TS \cdot (\forall p \in PRE \cdot p(t)) \wedge (\forall p \in POST \cdot p(t))$.
- The *invalid* mode ignores the postcondition: $\forall t \in TS \cdot \forall p \in PRE \cdot \neg p(t)$.
- The *checkpost* mode checks counterexamples of the postcondition, or execution results judged to be wrong: $\forall t \in TS \cdot (\forall p \in PRE \cdot p(t)) \wedge (\forall p \in POST \cdot \neg p(t))$.

[5] The JSTN language also provides syntax to explicitly declare which variables the target method reads or writes.

[6] JSTN uses the symbol ' to refer to post-state variables

Test design is additionally taken into consideration for $TS$. Suppose that $PART = \{part_1, \ldots, part_i\}$ and $CS = \{cs_1, \ldots, cs_j\}$ are sets of given partition predicates and test case predicates, respectively.

- The *valid* mode generates $TS$ such that $\forall p \in PART \cup CS \cdot (\exists t \in TS \cdot p(t))$. This means partitions and test cases have the same effect, though human engineers probably want to distinguish instances and partitions (classes) of instances. For the *invalid* and *checkpost* modes, there are identical elements for partitions and test cases but we do not repeat the definitions.
- The *checktest* mode is used with *valid* and *invalid*. For the *valid* run, the *checktest* mode adds $\forall t \in TS \cdot (\exists cs \in CS \cdot cs(t))$. This means only the given test cases are used, and the tool run succeeds only if they cover all the partitions (i.e., checking the conformance of the test cases to the test designs).

Assertions can also be defined.

- It is possible to declare each test case as deterministic: $DET \subset CS$. A check command can be executed in order to try to find counterexamples: $TS$ such that $|\exists det \in DET \cdot (\{t \in TS | IN(det)(t)\}| > 1)$. $IN(det)$ is a predicate that constrains the input in the same way as *det* but not constrain the output[7].
- For each test case it is possible to define the partitions it belongs to: $BELONG \in CS \times PART$. A check command can be executed to try to find counterexamples: $TS$ such that $\forall t \in TS \cdot (\exists (cs, part) \in BELONG \cdot cs(t) \wedge \neg part(t))$.

### 3.4 Prototype Implementation

The current implementation of the JSTN tool simply wraps Alloy Analyzer [1]. Alloy provides a primitive specification language for modeling sets and relations together with constraints in the first-order logic, to be used for model finding by a SAT solver. The JSTN tool uses the Xtext/Xtend frameworks [3] to extend the Java syntax, generate language tools including a parser and an editor on Eclipse, and implements the conversion from JSTN to Alloy. During the conversion, the tool keeps mapping between tags given by the engineer and names encoded in Alloy so that it can recover the tags from the solution.

Figure 2 shows the result view of the tool. Test cases are presented with tags in a table. The pulldown on the right allows the user to choose a test case to keep, which then shows a `case` description to be copied and inserted. Other feedback is possible, such as editing the presented values (to modify and keep test cases), and creating assertions by selecting wrong tags or indicating missing tags (to see problems on tags are resolved after modification and rerun of the tool).

Although JSTN applies the familiar Java syntax, it is intended to work as a modeling language, or validation of code by introducing by some bounds; it

[7] The current implementation simply limits the predicate to a conjunction of input constraints and output constraints when a test case is declared to be deterministic.

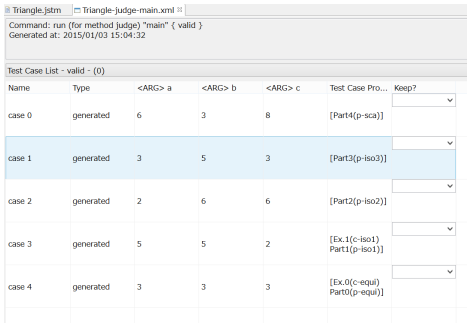


**Fig. 2.** Screenshot of JSTN Tool

is not intended to cover all the code-level details. The current implementation deals with primitive and standard object types of boolean, integer, char, String, Set, List and Map as well defined classes and enum types. The language provides operators common in specification languages, especially on Set, List and Map. For example, an operator `inds` is provided for obtaining a set of indexes for a list, together with a choice of Java method styles (`list.indexSet()`).

In addition to the core syntax elements in Section 3.3, the current implementation includes heuristics, options and syntax sugars. For example, the tool provided options to include well-known test design such as boundaries or forcing each test case for invalid input to only include one error. The tool also provided heuristics to automatically add partitions to include cases that satisfy the left side of implication and to use all the possible values when the type is boolean or enum (as well as the option to deactivate).

Primary issue in model finding is the inevitability of a state explosion for types that have enormous number of possible values. The default way that Alloy Analyzer works is to limit the number of instances for each signature (type of instances), the bit length of the integer values, or the size of list values. The JSTN tool allows to additionally set bounds on sets, that is, JSTN-level set values (not Alloy-level at which everything is a set). Although the JSTN language provides options to control these bounds, the tool basically tries to find sufficient sizes by analyzing the number of test cases required by partitions and examples, the used integer constant values, and so on.

The current implementation also uses symbolic values for the integer and string types, when the constraints on them contain only comparison operators (not arithmetic operators on integers or operators that accesses characters inside strings). For example, if an integer variable has a precondition `0 <= x && x <= 30`, set values are created such as $xNonboundary = [1, 29]$, $xUpperboundary = [30, 30]$, and $xOverNonboundary = [32, *]$. There can be multiple ranges, e.g., by made partitions, and relations between ranges (disjoint, contains, etc.) are analyzed and encoded to avoid the occurrence of impossible values (e.g., belong to both $[0, 10]$ and $[20, 30]$). After the model finding, values to be presented

are generated from the symbol values back to the constant values or arbitrarily generated within constraints, by using uniform distributions and string dictionaries. Note that in order to judge a variable can be symbolized or not, we need an iterative propagation algorithm. Suppose there are two formulas `x == y` and `x % 3 == 2`. If we only have the first formula, we can symbolize both `x` and `y`. However `x` is not symbolized due to the second formula, and then `y` is not, either, for the type consistency in the first formula.

The objective of the prototype implementation was to have experiences with industry people (shown in Section 4). It leaves potentials for performance improvement, e.g., by incorporation of a SMT solver.

## 4 Seminar Experiences

### 4.1 Overview

We held a one-day seminar to obtain feedback from engineers in industry as well as to evaluate the JSTN language and tool[8]. In the call for participation, we introduced the seminar as:

- Overviews three areas of formal specification, test-driven development and test design for quality assurance, discussing mutual relationships
- Uses a prototype tool wrapping a model finder (solver) for exercises to validate specifications by test cases, and to validate test cases by specifications.

The call was distributed to mailing lists of attendees of past seminars, primarily on formal methods, as well as those of our educational program for industry (lasting almost ten years) [10, 11]. It was also distributed through microblogs and social networks.

The seminars were held twice and drew in total 60 participants. As we put high priority on exercises in the limited time, we asked the participants to fill out questionnaires after the seminar. We got answers from 40 of them, including 34 from industry (the others were graduate students and academic researchers).

We asked about their previous experiences with formal methods and ASD/TDD, in four levels as shown in Table 1. Most of the participants had learned or had used formal methods. This is probably because there have been a lot of seminars, guide documents, etc. on formal methods in these several years. The rate of participants using ASD/TDD at work was also high, maybe due to the redistribution of the call by influential persons in these communities.

The seminar program consisted of four parts: introduction, formal specification, test-driven development, and test design for quality assurance. The introduction was similar to Section 3.1, though it had a simpler example. In the other three parts, we introduced general principles and gave an example of existing frameworks or tools. Then we introduced the related syntax and tool functions

[8] The call for participation and materials can be found on the following pages (in Japanese): `http://topse.or.jp/2014/11/2243` `http://research.nii.ac.jp/~f-ishikawa/jstm/`

**Table 1.** Experiences of Participants

| Item | Formal Methods | ASD/TDD |
|---|---|---|
| Using at work | 7 | 19 |
| Have learn but not using at work | 23 | 15 |
| Interested but have not learned or used | 7 | 4 |
| Thought irrelevant or did not know | 3 | 2 |

of JSTN and had the participants complete two exercises. The exercises were very short, e.g., writing a postcondition for the list reverse function and writing test cases for the triangle function for TDD.

### 4.2 Overall Evaluations

We asked about the advantages of the JSTN language and tool, providing choices and allowing for multiple choices. The results are shown in Table 2 (a). The feedback was very positive for the tool design discussed in Sections 3.1 and 3.2.

Similarly, we asked about the possible effective usages of the JSTN language and tool, as shown in Table 2 (b). Here, each engineer could agree with a few of more of the presented effective usages across the three target areas of the seminar. It is notable JSTN is considered effective to support TDD, together with the fact many participants use it at daily work. It is also notable that the seminar, by covering different areas, gave very good impressions on its educational value. The last item in the table (easy use of solvers) does not make a good sense as we could only overview such a usage in the seminar, e.g., deriving a system configuration that satisfies some constraints.

We also asked about the improvements that they deemed most important, as shown in Table 2 (c). The performance was the most critical problem. Some of the exercises in the seminar used arithmetic (e.g., the triangle function) and took a long time (tens of seconds) depending on the given description. There were also many opinions about practical aspects of interfacing, both between the tool and the user, as well as between the tool and other popular tools. The item "Others" included documentation as well as combination with other techniques such as automated random testing or bounded model checking. The target scope of JSTN, i.e., analyzing a (model of) function with generic model finding functionalities, seemed to be acceptable.

## 5 Discussion

### 5.1 Related Work

In summary, novelty of this paper is first we took an approach to allow for balance and mixture of automated and manual effort, focusing on continuous

**Table 2.** Evaluations

(a) Advantages

| | |
|---|---|
| Enable to incorporate principles from TDD, formal methods, test design | 22 |
| Enable to run the tool as soon as some description is given | 22 |
| Make easier use of formal methods and solvers | 17 |
| Provide opportunities to enlarge and learn viewpoints | 16 |
| Is a general-purpose tool to support various tasks to some extent | 14 |
| Enable lightweight usages such as partially automated test design | 14 |

(b) Effective Usages

| | |
|---|---|
| Education and enhancement of understanding and awareness to enlarge insights of mid-level engineers | 15 |
| Management and discussion on test cases in TDD | 14 |
| Introduction and education of foundations for beginners | 12 |
| Clarification and validation of constraints in domain analysis or specification construction | 11 |
| Assistance of test design for quality assurance | 10 |
| Clarification and validation of logic regardless of the task | 9 |
| Machine-readable standard comment formats on program code | 7 |
| Formalization and validation before using existing formal methods | 7 |
| Easy use of solvers on complex problems | 4 |

(c) Expected Improvements

| | |
|---|---|
| Performance (time to obtain a solution) | 18 |
| User interface (auto completion, tool-tips, etc.) | 14 |
| Conversion from/to common formats such as UML, Excel, testing frameworks | 8 |
| Others | 8 |
| Extension of target problems (e.g., execution sequence, component composition) | 3 |
| Specialization of functions and user interface for specific usages | 3 |
| Tuning on the number and variety of test cases | 2 |

specification and validation by human engineers. Second, we evaluated the tool with engineers in industry and obtained positive feedbacks not only on individual usages such as supporting TDD but also on effectiveness for education and enlightenment.

We discussed the necessity to test postcondition functions in Section 2.1. Although tools for VDM (VDMTools and Overture) internally generate such functions, no direct support of comprehensive testing is provided. A recent update in the VDM language (VDM-10) introduced syntax for a kind of test design using regular expressions (called "traces"). However this design is for generating a comprehensive set of test cases, and it takes long time to run all of them and is difficult to understand the meanings and results of them. The work presented here, on the other hand, while similarly aimed at test design, is intended to help engineers derive, or assert existence of, test cases for the sake of confidence.

Alloy originally focused on model finding [1], but in a primitive, not task-oriented way (discussed in Section 2.1). The presented work discussed a higher-level language and tool for direct support of tasks involved in specifications or test cases, as well as validation patterns with the nondeterministic nature of specification. Although the implementation is just a wrapping of Alloy, it allows for various automated tuning and heuristics during the conversion. For example, it allowed for automated implementation of symbolic representations in Alloy, which were only discussed theoretically in [19]. The study in [14] discussed application of TDD to Alloy by introducing syntax for partial instance. The presented work extended the direction by introducing a more high-level syntax and evaluating it with engineers. Aluminum is an extension Alloy that first presents minimum models (test cases in this paper) and then incrementally move to more complex ones [16]. In that study, "incrementally" means the addition of a relation and thus differs from the task-oriented meaning used in this paper. Aluminum is said to be effective in the initial phases where engineers have little confidence about specifications, not for validation or finding unexpected situations. Although the presented work had conceptually different directions from those studies, our future work will include improvement of the implementation by incorporating their techniques.

TestEra is a tool that uses Alloy to generate input of the test cases by preconditions and to assert their output by postconditions [12]. Using Alloy without test designs can easily lead to redundancy, or limited test performance (e.g., code coverage). Generally, results from automated test generation are too difficult for human engineers to understand and have confidence about them, or are limited in the test performance. There have been also a lot of studies on automated generation and execution of test cases with no or little human effort on test designs or specifications. Although the focus is different, our future work will include applications of the techniques used in such studies, e.g., recording and reusing generated objects to obtain more complex objects of larger classes [17].

Discussions on these kinds of tools for education and enlightenment have been very limited, while this study showed positive feedbacks by industrial engineers.

### 5.2 Limitations and Future Work

Further improvement and evaluation are necessary in terms of tool implementation. The future work includes combination of different kinds of solvers (including not only SMT solvers but also generation of values directly by the programming language, currently Java), as well as heuristics on problem decompositions. As the primary approaches depend on heuristics, it is necessary to have intensive evaluations with various problems. Further evaluation on usages and benefits of JSTN are also necessary, as the seminar was limited to short time with very small exercises. The long-term future work includes making JSTN into a framework by extracting language-independent parts so that implementations can be obtained for various specification languages.

## 6 Concluding Remarks

We presented a language and tool called JSTN that mixes specification, test design and test cases together with a model finder. This approach enables interactive, quick feedback as well as a flexible blend of different principles. The seminar yielded positive feedback from industry engineers about the potential benefits of the proposed language and tool, in addition to effectiveness on education and enlightenment. Given that this is the first step, we will continue improving the tool implementation as well as investigating educational experiences.

## Acknowledgment

This work was partially supported by the RISE program of IPA (Information-Technology Promotion Agency) in Japan. The authors would like to thank the students and researchers who gave us so much helpful advice, support for the implementations, and assistance in the sample development. The authors also thank all the engineers who attended the seminar and gave us a lot of feedback.

## References

1. Alloy. `http://alloy.mit.edu/alloy/`
2. Cucumber - making bdd fun. `http://cukes.info/`
3. Xtext - language development made easy! `http://www.eclipse.org/Xtext/`
4. Adzic, G.: Specification by Example: How Successful Teams Deliver the Right Software. Manning Pubns Co (2011)
5. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional (2002)
6. Black, S., Boca, P.P., Bowen, J.P., Gorman, J., Hinchey, M.: Formal versus agile: Survival of the fittest. IEEE Computer 42(9), 37–45 (September 2009)
7. Cok, D., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Device, Lecture Notes in Computer Science, vol. 3362, pp. 108–128 (2005)
8. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer (2005)
9. Freeman, S., Pryce, N.: Growing Object-Oriented Software, Guided by Tests. Addison-Wesley Professional (2009)
10. Honiden, S., Tahara, Y., Yoshioka, N., Taguchi, K., Washizaki, H.: Top SE: Educating Superarchitects Who Can Apply Software Engineering Tools to Practical Development in Japan. In: The 29th International Conference on Software Engineering (ICSE 2007). pp. 708–718 (2007)
11. Ishikawa, F., Taguchi, K., Yoshioka, N., Honiden, S.: What Top-Level Software Engineers Tackles after Learning Formal Methods - Experiences from the Top SE Project. In: The 2nd International FME Conference on Teaching Formal Methods (TFM 2009). pp. 57–71 (November 2009)

12. Khalek, S.A., Yang, G., Zhang, L., Marinov, D., Khurshid, S.: TestEra: a tool for testing java programs using alloy specifications. In: The 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). pp. 608–611 (November 2011)
13. Kuhn, D.R., Kacker, R.N., Lei, Y.: Practical combinational testing. Tech. Rep. NIST Special Publication 800-142, National Institute of Standards and Technology (October 2010)
14. Montaghami, V., Rayside, D.: Extending alloy with partial instances. In: Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012). pp. 122–135 (June 2012)
15. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, 3rd edn. (2011)
16. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: The 2013 International Conference on Software Engineering (ICSE 2013). pp. 232–241 (May 2013)
17. Pacheco, C., Ernst, M.D.: Randoop: Feedback-directed random testing for Java. In: The 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA 2007). pp. 815–816 (October 2007)
18. Polikarpova, N., Furia, C.A., Pei, Y., Wei, Y., Meyer, B.: What good are strong specifications? In: The 2013 International Conference on Software Engineering (ICSE 2013). pp. 262–271 (2013)
19. Siddiqui, J.H., Khurshid, S.: Symbolic execution of alloy models. In: The 13th International Conference on Formal Engineering Methods (ICFEM 2011). pp. 340–355 (October 2011)
20. Zimmerman, D.M., Nagmoti, R.: JMLUnit: The Next Generation. In: Revised Selected Papers from International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010). Lecture Notes in Computer Science, vol. 6528, pp. 183–197 (2011)