# Formal development of a test case generation tool

Gilbert Laycock

`Time-stamp: <03/07/17 16:58:22 gtl1>`
Last LaTeX-ed: 17th July 2003

## 1 Test specifications

A test specification under Ostrand and Balcer's Category-Partition method [?] is normally written out in their Test Specification Language (TSL). Ostrand and Balcer did not explicitly define their language, but a BNF can be constructed that matches the examples given in their paper, see [?]. In this paper I want to show how their TSL can be extended and simplified, and then specify and develop a tool that can be used to generate test frames from a test specification.

### 1.1 Limitations of Ostrand & Balcer's TSL

- Simple comments; I want to have "literate" test specifications which document themselves. This will mean that free text can be interleaved within the formal test specs.
- Distinction between "parameter" and "environment" categories in the specification. This is compounded by referring to the categories as "parameters" in the TSL.

## 2 Revised TSL BNF

My aim is to re-write the syntax so that it overcomes both these problems.

| | |
|---|---|
| , | the concatenate symbol |
| ::= | the define symbol |
| $\mid$ | the definition separator symbol |
| *[ ]* | enclose optional syntactic items |
| **{ }** | enclose syntactic items which may occur zero or more times |
| ‘ ’ | single quotes enclose terminal symbols |
| **Name** | non-terminal symbols written in **sans-serif** font |
| ; | terminator symbol indicating the end of a rule |

Figure 1: Special symbols used in BNF definitions.

### 2.1 Free text commentary

It is my intention that free text can be introduced into a specification in any way you like. Trying to build a syntax that allows for "commentary" in lots of different places would be rather intricate. Instead

I will adopt the approach that the literate specification is in a very simple meta-language that consists of free text and code interleaved. This meta-language is then pre-processed to strip out everything but code, and the results can then be passed on as raw TSL code. TSL code cannot be parsed until it has been stripped out of the literate specification.

The literate test specification consists of free text and code mixed together. Code is identified by simple mark-ups.

```
LitSpec   ::= [FreeText], {TSLChunk};
TSLChunk  ::= TSLCode, [FreeText];
TSLCode   ::= '\begin{code}',
                 TSLCodeBody,
              '\end{code}';
```

## 2.2 The TSL proper

Firstly, a test specification can specify the tests for several functional components of a system. Each of the components can be considered individually if required.

```
TestSpec ::= {FuncSpec};
```

Functional components are labelled consecutively with capital letters, and have a brief, unique, identifying name.

The tests for each functional component are described as a series of *category* specifications. A category corresponds to some characteristic feature of the functional component.

```
FuncSpec ::= Capital, ' Function:', FuncName, '\n',
               {CategorySpec},
             'EndFunction;\n';
```

Categories are numbered consecutively within each functional unit, and have a brief, unique, identifying name. A category is then described as a series of mutually exclusive *choices*. A choice describes a property of the category.

```
CategorySpec ::= Number, CategoryName, '\n',
                 {Choice};
```

Each choice has a brief descriptive identifying name, and can have any or all of

1. a property declaration;
2. a constraint (described in terms of other choices' properties)[1];
3. a flag (indicating that the choice is an "error" condition, or should only appear in a single test case.

```
Choice     ::= '*', ChName, [Property], [Constraint], [Flag];
Property   ::= '[property', PropName, ']';
Constraint ::= '[if', Condition, ']';
Flag       ::= '[error]'
             | '[single]';
Condition  ::= PropName
             | 'not(', Condition, ')'
             | '(', Condition, ')', LogOp, '(', Condition, ')';
```

[1] Ostrand and Balcer's constraints were simple conjunctions of properties; I have extended the language to allow any Boolean expression of properties.

There are a few odds and ends that still need to be defined.

$$
\begin{array}{lcl}
\textsf{\textbf{LogOp}} & ::= & \texttt{`and'} \\
 & | & \texttt{`or'}; \\
\textsf{\textbf{FuncName}} & ::= & \textsf{\textbf{Identifier}}; \\
\textsf{\textbf{CategoryName}} & ::= & \textsf{\textbf{Identifier}}; \\
\textsf{\textbf{ChName}} & ::= & \textsf{\textbf{Identifier}}; \\
\textsf{\textbf{PropName}} & ::= & \textsf{\textbf{Identifier}}; \\
\textsf{\textbf{Number}} & ::= & \textsf{\textbf{Digit}}, \{\textsf{\textbf{Digit}}\}; \\
\textsf{\textbf{Identifier}} & ::= & \textsf{\textbf{Letter}}, \{\textsf{\textbf{Letter}} \mid \textsf{\textbf{Digit}} \mid \textsf{\textbf{Punctuation}}\}; \\
\textsf{\textbf{Digit}} & ::= & \texttt{`0'} \mid \texttt{`1'} \mid \ldots \mid \texttt{`9'}; \\
\textsf{\textbf{Letter}} & ::= & \textsf{\textbf{Capital}} \\
 & | & \textsf{\textbf{Lower}}; \\
\textsf{\textbf{Capital}} & ::= & \texttt{`A'} \mid \ldots \mid \texttt{`Z'}; \\
\textsf{\textbf{Lower}} & ::= & \texttt{`a'} \mid \ldots \mid \texttt{`z'}; \\
\textsf{\textbf{Punctuation}} & ::= & \texttt{`\ '} \mid \texttt{`-'} \mid \texttt{`\_'};
\end{array}
$$

# 3 Test frame generation

A test frame is a description of a test case. It is simply a list of one choice from each category. The choices in a particular test frame must be consistent; that is, it must be possible to satisfy all of the constraints at the same time.

Each test frame has an identifier consisting of the letter that identifies the functional component, plus a number that is unique within that functional component.

The idea is that test frames can be trivially converted into actual test cases and test scripts. Test cases describe the precise inputs required, and also describe the acceptable outputs for those inputs. A test script will also put the test cases into a suitable order, and include any information required to set up and execute the test cases (*e.g.* compiler options, path settings *etc.*).

## 3.1 Test frame syntax

Each test frame consists of a label, followed by a list of choices.

$$
\begin{array}{lcl}
\textsf{\textbf{TestFrame}} & ::= & \textsf{\textbf{Label}}, \texttt{`\textbackslash n'} \\
 & & \quad \{\textsf{\textbf{FChoice}}\};
\end{array}
$$

A choice is here simply the name of one of the choices from the test specification.

$$\textsf{\textbf{FChoice}} ::= \texttt{`*'}, \textsf{\textbf{ChName}}, \texttt{`\textbackslash n'};$$

A label is a letter, number combination.

$$\textsf{\textbf{Label}} ::= \textsf{\textbf{Capital}}, \texttt{`.'}, \textsf{\textbf{Number}};$$

Test frames are combined together to make a test description for functional components.

$$
\begin{array}{lcl}
\textsf{\textbf{TestDescription}} & ::= & \textsf{\textbf{Letter}}, \texttt{`.\ \ \ '}, \textsf{\textbf{FuncName}}, \texttt{`:\textbackslash n'}, \\
 & & \quad \textsf{\textbf{TestFrame}}, \{\textsf{\textbf{TestFrame}}\} \\
 & & \texttt{`EndFunction \textbackslash n'};
\end{array}
$$

Test descriptions for several functional components can be collected together into a test suite description.

$$\textsf{\textbf{TestSuiteDescription}} ::= \{\textsf{\textbf{TestDescription}}\};$$

# 4 Generator specification

In sections 2 and 3 the precise syntax of both test specifications and test frames was laid out. I now want to specify exactly what the relationship between the two is. I will be using the Z notation [?] for this.

The crucial relationship is between the test specification for one functional component and the corresponding collection of test frames.

## 4.1 A category *Choice*

A test specification for a functional component is composed (at the lowest level) of *Choice*s. *Choice*s are made up from a few components, some of which come straight from the abstract syntax, such as choice names and property names.

$$[\mathsf{ChName}, \mathsf{PropName}]$$

Other components need redefining. A choice may have a property,

$$\begin{aligned} PROPERTY &::= nullP \\ &\quad\;\; |\;\; prop\langle\!\langle \mathsf{PropName} \rangle\!\rangle \end{aligned}$$

and/or a constraint

$$\begin{aligned} CONSTRAINT &::= nullC \\ &\quad\;\; |\;\; cond\langle\!\langle CONDITION \rangle\!\rangle \\ CONDITION &::= propC\langle\!\langle \mathsf{PropName} \rangle\!\rangle \\ &\quad\;\; |\;\; notC\langle\!\langle CONDITION \rangle\!\rangle \\ &\quad\;\; |\;\; andC\langle\!\langle CONDITION \times CONDITION \rangle\!\rangle \\ &\quad\;\; |\;\; orC\langle\!\langle CONDITION \times CONDITION \rangle\!\rangle \end{aligned}$$

I will want to be able to compare a particular *PROPERTY* with a particular *CONSTRAINT*.

$$\begin{array}{|l} EqProperty : PROPERTY \times CONSTRAINT \rightarrow \mathbb{B} \\ \hline \forall\, p : PROPERTY \bullet \forall\, c : CONSTRAINT \bullet \\ \quad EqProperty(p, c) \Leftrightarrow ((p = nullp) \wedge (c = nullC)) \vee \\ \qquad\qquad (\exists\, n : \mathsf{PropName} \bullet \\ \qquad\qquad\quad ((p = prop(n)) \wedge (c = cond(propC(n)))) \end{array}$$

I want to be able to extract the names of all of the properties involved in a constraint. This is simply a matter of constructing the set of all **PropName**s used in the *CONSTRAINT*.

$$\begin{array}{|l} inConstraint : CONSTRAINT \rightarrow \mathbb{F}_1\, \mathsf{PropName} \\ inCondition : CONDITION \rightarrow \mathbb{F}_1\, \mathsf{PropName} \\ \hline inConstraint(c) = \mu\, ps : \mathbb{F}_1\, \mathsf{PropName} \mid \\ \qquad (ps = \varnothing \wedge c = nullC) \vee \\ \qquad (\exists\, con : CONDITION \bullet \\ \qquad\quad (ps = inCondition(con) \wedge c = cond(con))) \\ \forall\, c : CONDITION;\ p : \mathsf{PropName};\ c_1, c_2 : CONDITION \bullet \\ \quad c = propC(p) \Rightarrow inCondition(c) = \{p\} \\ \quad c = notC(c_1) \Rightarrow inCondition(c) = inCondition(c_1) \\ \quad c = andC(c_1, c_2) \Rightarrow inCondition(c) = inCondition(c_1) \cup inCondition(c_2) \\ \quad c = orC(c_1, c_2) \Rightarrow inCondition(c) = inCondition(c_1) \cup inCondition(c_2) \end{array}$$

I will also be wanting to "evaluate" constraints within a particular "context". A context will simply be a collection of property names.

$$CONTEXT == \mathbb{F}_1\ \textsf{PropName}$$

A *nullC CONSTRAINT* is always satisfied, *i.e.* evaluates to true. Other *CONSTRAINT*s are defined in terms of *CONDITIONS*. A "simple" condition, consisting of simply a **PropName** is evaluated by checking whether or not that property exists in the context. If it does, the property evaluates true, otherwise false. More complex conditions are evaluated recursively.

$$
\begin{array}{l}
EvalConstraint : CONSTRAINT \times CONTEXT \to \mathbb{B} \\
EvalCond : CONDITION \times CONTEXT \to \mathbb{B} \\
\hline
\forall c : CONSTRAINT;\ ctxt : CONTEXT \bullet \\
\quad (c = nullC \land EvalConstraint(c, ctxt) = \mathrm{true}) \\
\quad \lor \\
\quad (\exists con : CONDITION \bullet \\
\qquad c = cond(con) \land EvalConstraint(c, ctxt) = EvalCond(con, ctxt)) \\
\forall con : CONDITION;\ ctxt : CONTEXT \bullet \\
\quad \exists c_1, c_2 : CONDITION;\ p : \textsf{PropName} \bullet \\
\quad con = propC(p) \Rightarrow EvalCond(c, ctxt) = p \in ctxt \\
\quad con = notC(c_1) \Rightarrow EvalCond(c, ctxt) = \neg\ EvalCond(c_1, ctxt) \\
\quad con = andC(c_1, c_2) \Rightarrow EvalCond(c, ctxt) = EvalCond(c_1, ctxt) \land EvalCond(c_2, ctxt) \\
\quad con = orC(c_1, c_2) \Rightarrow EvalCond(c, ctxt) = EvalCond(c_1, ctxt) \lor EvalCond(c_2, ctxt)
\end{array}
$$

Finally, a choice can have a flag

$$
\begin{array}{lll}
FLAG & ::= & nullF \\
 & | & singleF \\
 & | & errorF
\end{array}
$$

I can now define what a *CHOICE* actually is. The only invariant condition is that a *CHOICE*'s constraint cannot be defined in terms of its own property.

$$[CH\_ID]$$

$$
\begin{array}{l}
\underline{\quad CHOICE \quad} \\
id : CH\_ID \\
ChN : \textsf{ChName} \\
prop : PROPERTY \\
constraint : CONSTRAINT \\
flag : FLAG \\
\hline
prop = nullP \lor prop \notin inConstraint(constraint)
\end{array}
$$

## 4.2 A category

Categories are built up from a collection of choices. Within a category choices can be identified by unique identifiers, so I will use a partial function to model them.

There are several invariants on categories:

1. Each choice must have a unique name.

2. Each property defined must have a unique name.
3. No constraint can be defined in terms of other properties of **this** category.
4. At least one choice must be "unflagged".

$[CAT\_ID]$

$$
\begin{array}{l}
\textit{CATEGORY} \\
\hline
id : CAT\_ID \\
catN : \textsf{\textbf{CategoryName}} \\
Choices : CH\_ID \rightarrowtail CHOICE \\
\hline
\nexists ch_1, ch_2 : CHOICE \bullet \\
\quad ch_1, ch_2 \in \operatorname{ran} Choices \\
\quad ch_1 \neq ch_2 \\
\quad ch_1.ChN = ch_2.ChN \\
\nexists ch_1, ch_2 : CHOICE \bullet \\
\quad ch_1, ch_2 \in \operatorname{ran} Choices \\
\quad ch_1 \neq ch_2 \\
\quad ch_1.prop \neq nullP \land ch_1.prop = ch_2.prop \\
\nexists ch_1, ch_2 : CHOICE \bullet \\
\quad ch_1, ch_2 \in \operatorname{ran} Choices \\
\quad ch_1 \neq ch_2 \\
\quad ch_1.prop \in inConstraint(ch_2.constraint) \\
\exists\, ch : CHOICE \bullet \\
\quad ch \in \operatorname{ran} Choices \\
\quad ch.flag = nullF \\
\hline
\end{array}
$$

## 4.3 A functional component's test specification

A functional component has a letter to identify it, a unique name and is then made up of a (non-empty) sequence of categories.

There are several invariants on the categories:

1. Each category has a unique name.
2. No two choices in any two categories have the same property.
3. All of the constraints defined are defined in terms of properties of choices in this functional component.

$$
\begin{array}{l}
\textit{FUNC\_TEST\_SPEC} \\
\hline
id : \textsf{Capital} \\
funcN : \textsf{FuncName} \\
Categories : CAT\_ID \nrightarrow CATEGORY \\
\hline
\nexists cat_1, cat_2 : CATEGORY \bullet \\
\quad cat_1, cat_2 \in \mathrm{ran}\ Categories \\
\quad cat_1 \neq cat_2 \\
\quad cat_1.catN = cat_2.catN \\
\nexists ch_1, ch_2 : CHOICE \bullet \\
\quad \exists\ cat_1, cat_2 : CATEGORY \bullet \\
\qquad cat_1, cat_2 \in \mathrm{ran}\ Categories \\
\qquad cat_1 \neq cat_2 \\
\qquad ch_1 \in \mathrm{ran}\ cat_1.Choices \\
\qquad ch_2 \in \mathrm{ran}\ cat_2.Choices \\
\qquad ch_1.prop \neq nullP \wedge ch_1.prop = ch_2.prop \\
\forall\ c : CONSTRAINT \bullet \\
\quad c \in \{cat : CATEGORY;\ ch : CHOICE \mid \\
\qquad\qquad cat \in \mathrm{ran}\ Categories \\
\qquad\qquad ch \in \mathrm{ran}\ cat.Choices \\
\qquad \bullet\ ch.constraint\} \\
\quad inConstraint(c) \subseteq \{cat : CATEGORY;\ ch : CHOICE \mid \\
\qquad\qquad\qquad\qquad cat \in \mathrm{ran}\ Categories \\
\qquad\qquad\qquad\qquad ch \in \mathrm{ran}\ cat.Choices \\
\qquad\qquad\qquad\qquad ch.prop \neq nullP \\
\qquad\qquad\qquad \bullet\ ch.prop\}
\end{array}
$$

# 5 Test frame generation

Test frames can be generated systematically from a function test specification.

## 5.1 Properties of each test frame

A test frame only makes sense in the context of a particular function component test specification.

There are two types of test frame, “flagged” and “unflagged”.

**Unflagged:** Each test frame must satisfy the following conditions:

1. Must be made up of exactly one choice from each of of the function’s categories.
2. Must not include any “flagged” choices.
3. Must satisfy all of its constraints. That is, the constraint on each choice must evaluate true in the context of the properties of the other choices.

**Flagged:** Each test frame must satisfy the following conditions:

1. Is made up of precisely **one** choice (the *primary* choice) from amongst all of the categories’ choices.
2. The choice must be flagged.
3. Lists **only** those other choices (for other categories) that must be used to meet any constraint on the primary choice.

In addition, each frame has a unique, two part label. The first part is simply the identifying letter of the function; the second part is a number uniquely identifying the test frame among all those for this function.

$$
\begin{array}{l}
\textit{FRAME} \\
\hline
label : \mathsf{Capital} \times \mathbb{N}_1 \\
fr\_context : FUNC\_TEST\_SPEC \\
fr\_choices : CAT\_ID \twoheadrightarrow CH\_ID \\
flagged : \mathbb{B} \\
\hline
\exists\, n : \mathbb{N}_1 \bullet \\
\quad label = (fr\_context.id, n) \\
(flagged = \mathrm{true} \land \\
\quad \mathrm{dom}\, fr\_choices = \mathrm{dom}\, fr\_context.Categories \\
\quad \forall\, cat : CAT\_ID \bullet \\
\qquad cat \in \mathrm{dom}\, fr\_choices \\
\qquad fr\_context.Categories(cat).Choices(fr\_choices(cat)).flag = nullF \\
\qquad EvalConstraint(fr\_context.Categories(cat).Choices(fr\_choices(cat)).constraint, \\
\qquad\qquad \{cat' : CAT\_ID \mid cat' \in (\mathrm{dom}\, fr\_choices) \land cat' \neq cat \\
\qquad\qquad \bullet\, fr\_context.Categories(cat').Choices(fr\_choices(cat')).property\})) \\
\lor \\
(flagged = \mathrm{false} \land \\
\quad \mathrm{dom}\, fr\_choices \subseteq \mathrm{dom}\, fr\_context.Categories \\
\quad \exists\, cat : CAT\_ID \bullet \\
\qquad cat \in \mathrm{dom}\, fr\_choices \\
\qquad fr\_context.Categories(cat).Choices(fr\_choices(cat)).flag \neq nullF \\
\qquad \forall\, cat' : CAT\_ID \bullet \\
\qquad\quad cat' \in \mathrm{dom}\, fr\_choices \land cat \neq cat' \\
\qquad\quad EvalConstraint(fr\_context.Categories(cat').Choices(fr\_choices(cat')).constraint, \\
\qquad\qquad\quad \{cat'' : CAT\_ID \mid cat'' \in (\mathrm{dom}\, fr\_choices) \land cat'' \neq cat \land cat'' \neq cat' \\
\qquad\qquad\quad \bullet\, fr\_context.Categories(cat'').Choices(fr\_choices(cat'')).property\})) \\
\hline
\end{array}
$$

## 5.2 The set of test frames

The set of all possible test frames for a particular function is now simple to define. The final quantification is to ensure that distinct frames have distinct labels, and that no frame is duplicated with a different label.

$$
\begin{array}{l}
\textit{Generate\_test\_frames} \\
\hline
context? : FUNC\_TEST\_SPEC \\
tests! : \mathbb{F}\, FRAME \\
\hline
tests! = \{f : FRAME \mid f.fr\_context = context?\} \\
\forall\, f_1, f_2 : FRAME \bullet \\
\quad \{f_1, f_2\} \subseteq tests! \\
\quad (f_1.fr\_choices = f_2.fr\_choices) \Rightarrow (f_1 = f_2) \\
\hline
\end{array}
$$

# References