Using Haskell to Script Combinatoric Testing of Web Services

I.S.W.B. Prasetya, J. Amorim
Dept. of Inf. and Comp. Sciences
Utrecht University
Email: wishnu@cs.uu.nl

T.E.J. Vos, A. Baars
Dept. de Sist. Informaticos y Comp.
Univ. Polit. de Valencia
Emails: tvos@dsic.upv.es, abaars@pros.upv.es

Abstract— **The Classification Tree Method (CTM) is a popular approach in functional testing as it allows the testers to systematically partition the input domain of an SUT, and specifies the combinations they want. We have implemented the approach as a small domain specific language (DSL) embedded in the functional language Haskell. Such an embedding leads to clean syntax and moreover we can natively access Haskell's full features. This paper will explain the approach, and how it is applied for testing Web Services.**

Keywords- automated testing, combinatoric testing

I. Introduction

The Classification Tree Method (CTM) is a combinatoric testing approach proposed by Grochtmann and Grimm [2]. It allows a tester to specify how the domain of each parameter of a target function e.g. *f(x,y,z)* is abstractly partitioned into various subsets. The partitioning can be hierarchical; the overall partitioning has thus a tree structure and is called classification tree. The lowest level partitions are called 'classes'; each will be represented by a single concrete test-value. By combining classes of *x*, *y*, *z* we essentially construct test-cases. We can either generate all possible combinations, or select just some. The CTM approach can be used in both white box or black box testing, and has been successfully applied in many industrial projects [4]. There are tools like EXTRACT [9], ADDICT [1], or CTE-XL [4] that also provide visual editors to allow testers to visually create a classification tree, and specify the test-cases they want.

We will presents an implementation of CTM as a small Domain Specific Language (DSL). The DSL is called CTy, embedded in a functional language called Haskell. We will have to drop the luxury of a visual editor, but we get something else in return. Haskell is a very good host language for embedding DSLs. Successful examples include DSLs for writing financial contracts [3], parsers [8], and animation [5]. DSLs in Haskell gets native access to its features, e.g. clean syntax, lazy evaluation, polymorphism, and higher order functions. For us it means that we can write powerful expressions to declaratively specify the combinations of the classes that we want to include in our test-suite, to combine test-suites, and to sort the suites.

Essentially a web service is a function f that can be called over a network through a popular protocol such as HTTP. The function may have side effect, e.g. if it works on some persistent resource behind it. Parameters as well as f's reply are sent in an exchangeable format such as XML. These allow web services to be easily composed to build more complicated services. Because of their function-like behavior, the CTM approach can be directly used on them. In Section VI we will discuss how CTy can be used to combinatorically test a web service.

II. Specifying Input Domain

Imagine the function below (which can be provided as a web service) for calculating the premium's price of some insurance. Imagine we want to test it.

```
calcPremium(type,age,city)
```

where `type` is the type of insurance (e.g. standard, comfort, super, etc).

Rather than providing arbitrary test-cases, in the CTM approach a tester first specifies how the domain of each parameter of the function above will be partitioned. The partitioning is specified with a tree structure called *classification tree*. Figure 1 shows such a tree for `calcPremium`, expressed in CTy. E.g. it says that the parameter `age` is divided into three partitions: `kid1`, `kid2`, and `adult`. The last one is partitioned further into `young` and `old`.

Partitions of the lowest level (which are not further partitioned) are special, and are called *classes*. Each class logically specifies a subdomain. However, for testing purpose all values in this subdomain are considered equivalent. Under this assumption we can represent a class with just one test-value. This is the value we will use when the class is later used to form a test-case. We construct a class using the following notations:

```
"kid1" %= Int32 4
```

This constructs a class named `kid1`, with the value 4 as the concrete test-value representing it.

```
ctree1 = tree [insuranceType,age,place]
  where
  insuranceType = "insType" <== [
    "standard" %= frag Standard,
    "comfort" %= frag Comfort,
    "super"   %= frag Super ]
  age = "age" <== [
    "kid1" %= Int32 4,
    "kid2" %= Int32 16,
    adult ]
    where
    adult = "adult" <== [
      "young" %= Int32 20,
      "old"   %= Int32 66 ]
  place = "place" <== [
    "city"    %= StringN 100 "Delft",
    "noncity" %= StringN 100 "Achterhoek" ]
```

Figure 1. A classification tree in Haskell/CTy.

We will assume each class to have a unique name. Test-values are represented by a separate type `TestVal` that supports a number of constructors. E.g. `Int32` in the above example represents a 32-bits two's complement integer. In Haskell itself we can have integers of any size, but because we may target a different language then it is necessary to specify the specific range that we want. The constructor `StringN` *n s* is used to construct a string (s) whose length is at most *n*.

Non-class partitions which are also not top-level are represented by the type Partition, and are constructed using the following notation:

`"adult" <== [`$p_1, \dots, p_n$`]`

This constructs a partition named `adult`, with all those p_i's as subpartitions.

A top-level partition specifies the whole domain of a parameter of our target function. In the tradition of CTM this is called *category*. So, for the `calcPremium` example its classification tree will consists of three categories: one for the parameter `type`, one for `age`, and one for `city`. We will represent a category with its own type `Category`, but use the same `<==` notation as above to construct it.

Finally, a classification tree itself is represented by the type `CTree`, and is constructed with notation `tree [`C_1, ..., C_n`]`, where each C_i is a category. See also the example in Figure 1.

III. TEST-CASE AND TEST-SUITE

A *test-case* specifies an input for the function under the test, and the expected result (oracle). We will only address the problem of input generation. Oracles are either not needed (if we have executable specifications), or else cannot be in general automatically generated.

Since each class contains a test-value, a test-case can be represented by a list *t* of classes where every category (parameter) of `calcPremium` is represented in *t* by exactly one class. E.g. the list [city, standard, kid1] would represent a test-case for `calcPremium`. Concretely, this corresponds to these inputs:

```
calcPremium(Standard,4,"Delft")
```

We will however call any list of classes a 'test-case'. Those that satisfy the above criterion will be called 'complete' test-cases. We introduce these types to represent test-case and the corresponding notion of test suite :

```
type TestCase = [Class] --list of classes
type Suite = [TestCase] --list of test-cases
```

Below we define a number of auxiliary notations.

If *p* is a class or a partition, *cat*(*p*) denotes the category to which *p* belongs. If *P* is a list or a set of classes or partitions, we define $cat(P) = \{cat(p) \mid p \in P\}$. Since a test-case is also a list of classes, we can use the same notation on it. If φ is a classification tree, we will write *cat*(φ) to denote the set of categories in φ.

A test-case *t* is *well-formed* if for any distinct $c,d \in t$, $cat(c) = cat(d)$. With respect to a given classification tree φ, *t* is *complete* if it is well-formed and furthermore $cat(t) = cat(\varphi)$. A well-formed test-case that is not complete is called *partial*.

If *p* is a partition or a category, we write *class*(*p*) to denote the classes that are descendants of *p* (as a tree). In other words, these are the classes that make up *p*. If *P* is a list or set of partitions, $class(P) = \cup\{class(p) \mid p \in P\}$. We will also write $c \in p$ to mean $c \in class(p)$. Similarly, we also write $c \in P$.

Let *s* and *t* be test-cases. We write $s \supseteq t$ if as sets *s* subsumes *t*. We write $s \equiv t$ to mean that they subsume each other. The union and intersection of two suites are defined as follows:

$$S \cup T \;=\; S \mathbin{++} [\, t \mid t \in T, (\forall s \in S :: t \equiv s)\,]$$

$$S \cap T \;=\; [\, t \mid t \in T, (\exists s \in S :: t \equiv s)\,]$$

IV. SPECIFYING COMBINATIONS

Given a classification tree φ, we can generate its *full suite*, consisting of all possible test-cases it induces. We can also generate its *minimalistic suite*, which is the smallest suite such that every class in φ is covered at least once. Unfortunately, for a function with complex input domains, its full suite is often too large (e.g. thousands in size). On the other hand, the minimalistic suite is often too small.

Notice that a test-case can be seen as a combination over classes from different categories. Inspired by CTE-XL [4], in CTy we provide a simple language to express 'combination rules' to declaratively specify the combinations that we want to cover.

The combination rules are implemented 'symbolic'. It means that they are represented as data in Haskell and have to be interpreted first to actually produce the corresponding suites. This allows us to check them first for their well-formedness, before interpreting them. A combination rule will be represented by the type `Rule`, which can be constructed in this way:

$$R ::= \texttt{incl}\ P \ \mid\ R\ op\ R \ \mid\ \texttt{neg}\ R$$

where P is a so-called 'monocat'. It is either single a category name or a list of partition names, such that $|cat(P)| = 1$ (all partitions in P should belong to the same category). The op is one of these operators:

$$|{+}|,\quad |\&|,\quad \&\&*,\quad \&\&-$$

Semantically, a rule R specifies a suite over the categories mentioned in R. We will define a function `rule` R that interprets R and produces the suite it specifies. This function is essentially our test-cases generator. R may indeed produce test-cases which are still partial —we will deal with this later.

A suite S is well-formed if it consists of well-formed test-cases, and for all $t,u \in S$, $cat(t) = cat(u)$ (all test-cases in S cover exactly the same set of categories). Two well-formed suites can be combined more efficiently since we do not have to keep checking whether two test-cases from the suites are actually compatible to be combined. Combination rules will be restricted (and checked) so that they only produce well-formed suites.

The simplest rule is of the form incl P. Since P is a monocat, all classes under P belong to the same category. Each is a candidate test-value for P 's category. Each can be lifted to a partial test-case; and collecting them all form a suite. So:

$$\texttt{rule}(\texttt{incl}\ P) = [\,[c] \mid c \in class(P)\,]$$

This operator provides the union of two rules:

$$\texttt{rule}(R1\ |{+}|\ R2) = \texttt{rule}\ R1 \cup \texttt{rule}\ R2$$

Similarly, we define intersection $|\&|$. To be well-formed, $R1$ and $R2$ above should specify the same set of categories.

If $R1$ and $R2$ are rules with no common categories, this operator constructs the Cartesian product of their test-cases:

$$\texttt{rule}\ (R1\ \&\&*\ R2) = \texttt{rule}\ R1 \times \texttt{rule}\ R2$$

$$S1 \times S2 = [\, s1{+}{+}s2 \mid s1 \in S1,\ s2 \in S2\,]$$

The 'minus'-variant will combine $R1$ and $R2$ in a minimalistic way:

$$\texttt{rule}\ (R1\ \&\&-\ R2) = \texttt{rule}\ R1 + \texttt{rule}\ R2$$

$S1 + S2$ is defined as follows. If $|S2| < |S1|$, we fill $S2$ by duplicating some elements to make its size equal to $S1$; else we do the opposite. Then the definition looks as below:

$$S1 + S2 = [\, s_0{+}{+}\,t_0,\ \ldots,\ s_n{+}{+}\,t_n\,]$$

s_i's and t_i's are elements of respectively $S1$ and $S2$.

Suppose $N1$ and $N2$ are the size of rules $R1$ and $R2$. Whereas $\&\&*$ produces a suite of size $N1*N2$, that of $\&\&-$ has the size of $max(N1,N2)$.
The operator `neg` R constructs the 'complement' of R's suite. However this complement is not always calculated with respect to the full suite. Consider this example:

$$R = (\texttt{incl}\ [\text{standard}]\ \ \&\&*\ \ \texttt{incl}\ [\text{adult}])$$
$$\&\&-$$
$$\texttt{incl}\ [\text{city}]$$

`neg` R will complement R with respect to (`insuranceType` $\&\&*$ `age`) $\&\&-$ `place`. So, we take into account whether constituents of R are combined fully or minimally. The formal definition of `neg` can be found in our extended paper [6].

A. Derived operators

For further convenience we can define a whole array of derived operators, e.g. as below, based on the full combinator $\&\&*$. Let P be a monocat, and $R1$, $R2$ be rules that have no common category:

$$\texttt{excl}\ P = \texttt{neg}\ (\texttt{incl}\ P)$$

$$R1\ \texttt{or}_f\ R2 = (R1\ \&\&*\ R2)$$
$$|{+}|\ (\texttt{neg}\ R1\ \ \&\&*\ \ R2)$$
$$|{+}|\ (R1\ \ \&\&*\ \texttt{neg}\ R2)$$

$$R1\ \texttt{xor}_f\ R2 = (R1\ \&\&*\ \texttt{neg}\ R2)\ |{+}|\ (\texttt{neg}\ R1\ \ \&\&*\ \ R2)$$

$$R1\ \texttt{equ}_f\ R2 = (R1\ \&\&*\ R2)\ |{+}|\ (\texttt{neg}\ R1\ \ \&\&*\ \texttt{neg}\ R2)$$

$$R1\ \texttt{imp}_f\ R2 = \texttt{neg}\ R1\ \ \texttt{or}_f\ \ R2$$

Analogously we can have the minimalistic version of those operators, which are based on the combinator $\&\&-$.

V. COMBINING TEST-SUITES

Applying the function `rule` interprets a given combination rule R to produce the corresponding test-suite. Since suites are just lists, we can further combine and process them with any Haskell function of a compatible type. This can give us a lot of expressiveness. However, because we now directly combine suites (rather than rules), we will not check their well-formedness during the combinations. Doing so would be inefficient. We will just filter the final suite to throw away ill-formed test-cases. Since such filtering can make the meaning of suite operators less predictable, the user is now responsible to make sure that that does not happen (by not producing intermediate ill-formed test-cases).

We can combine two suites like this: $S1$ *op* $S2$, where *op* is one of the following operators:

|+|, |&|, `‘suchthat‘`, `‘except‘`

The first two are just the union and intersection over suites (the $\cup$ and $\cap$ we defined before). The arguments of these two operators should be suites over the same categories.

$S1$ `‘suchthat‘` $S2$ specifies a subset of $S1$ consisting of those test-cases t that subsumes some test-case in $S2$:

$$S1 \text{ ‘suchthat‘ } S2 \;=\; [\, s \mid s \in S1, (\exists t \in S2 :: s \sqsupseteq t)\,]$$

Then we can define the negative counterpart of this:

$$S1 \text{ ‘except‘ } S2 \;=\; S1 \;/\; (S1 \text{ ‘suchthat‘ } S2)$$

There are few more CTy operators that we do not discuss here, e.g. operators to specify k-wise combinations, to do relational join, and to sort a suite according to the weight of its test-cases. See our full paper [6].

A. Padding test-suite

Let f be the target function to test, and φ is the classification tree we want to use. A suite expression such as discussed above may not produce complete test-cases if the underlying combination rules in the expression do not mention all categories of φ. Before we can turn them to concrete test-cases for f, we need to pad/extend these test-cases to make them complete.

Let S be a test-suite. Suppose $\Delta = [D_1, \ldots, D_N]$ is the set of categories which are still missing from the test-cases in S. We can pad these test-cases by either fully or minimally combining them with Δ. This gives the following padding functions:

$$\text{paddingf}_{\varphi} S \;=\; S \times T_1 \times \ldots \times T_N$$

$$\text{paddingm}_{\varphi} S \;=\; S + T_1 + \ldots + T_N$$

where T_k = `rule(incl` D_k); $\times$ and $+$ are operators on test-suites as defined before.

B. Generating concrete suite

After padding (above) we will get a 'test-suite' that is ready to be turned into a concrete test-suite (which will drive the actual SUT). CTy provides a facility to do this. This facility generates a new Haskell program, that will actually execute the suite. However, this generated program does not contain oracles (which can't be generated). Instead, we just leave place holders for them. The tester has to fill these in manually.

VI. A BIGGER EXAMPLE: REGRESSING GOOGLE GEOCODING API

Google Geocoding API is a web-service provided by Google to calculate the geographic coordinate of a place. The place is described by its address, e.g. "`padualaan 14, utrecht, netherlands`". A coordinate is given in latitude and longitude e.g. (52.08503, 5.1704884). It is a RESTful service; we can query it by sending a HTTP request e.g. like this (for the above example):

```
http://maps.googleapis.com/maps/api/geocode/x
ml? address=padualaan+14+utrecht+netherlands
```

If the address is recognized, an XML containing its coordinate is returned. If it matches multiple places, the XML will contain multiple coordinates as well.

The service can handle different ordering of the components of an address (e.g. if we specify the town before the street), or if we mangle some component a little bit (e.g. '`utrecth`' instead of '`utrecht`'), or even omit the component. Note: there are cases where it seems to have problems.

Suppose we have a set P of important places/addresses (e.g. hospitals in your town). We want to make sure that over time Geocoding API always give consistent coordinates of those places. We will do this by regularly performing a regression test on the service. We use CTy to specify the test suite to use. We will construct one generic suite which can be instantiated for each address in P.

Let a be a full and correct address. We will construct a test-suite S_a where each test-case is obtained by permuting and mangling a's components in various ways. S_a is then used for regression with respect to a. This is done in two stages.

The first stage is a *manual approval stage*. For every test-case in S_a we verify if Geocoding API's answer is 'correct'. Every test-case is basically a query using some variant of a, sent to the API. The answer is correct if either the API says it doesn't recognize the query, or else if the coordinates returned by the API contains a correct one.

Once all test-cases in S_a are approved, we will store the API's answer of each test-case t and uses it as the oracle for t. Then we can proceed to the *regression stage* where we regularly re-run the S_a ; it will report an error if one query gives a different answer than the stored answer (the one we approved earlier). We will do this for every address a in our set P.

The API is tested through the test-interface below. The interface takes care of e.g. the HTTP connection with the API, formatting a query before it is sent to the API, and parsing the API's XML answer. Additionally it also: (1) mangles and permutes the given address according to the given mangling and order specifiers, and (2) checks the answer against the oracle.

`ggTI` *address coM ciM stM nrM o*

The function returns true if the result is equal to the oracle; else false. The parameter *address* is specified by a tuple (*country*, *city*, *street*, *streetnr*). The next four parameters are mangling-specifiers for the, respectively, *country*, *city*, *street*, and *streetnr* component of the given address. Finally *o* specifies the order/permutation of the address' components in the concrete query to the Geocoding API (e.g. `padualaan+14+utrecht`, or `utrecht+padualaan+14`).

We use a classification tree to express various ways to mangle each address component, and various permutations we want to consider; this is shown in Figure 2. Roughly for each address component we have four mangling modes: no-mangling, swapping the last two letters, dropping the last letter, and deleting the whole component. For street numbers, only the first and the last modes are used.

```
ct a  =  tree [ address,
                countryMode, cityMode,
                streetMode, numberMode,
                order ]
   where

   address = "Address" <== ["Address" %= addr_ a]

   countryMode = "CountryMode" <==
       [ "CoCorrect" %= mode_ Correct,
         "CoSwapped2" %= mode_ Swapped2,
         "CoDropLast" %= mode_ DropLast,
         "CoEmpty" %= mode_ Empty ]

   cityMode = "CityMode" <==
       [ "CiCorrect" %= mode_ Correct,
         "CiSwapped2" %= mode_ Swapped2,
         "CiDropLast" %= mode_ DropLast,
         "CiEmpty" %= mode_ Empty ]

   streetMode = "StreetMode" <==
       [ "StCorrect" %= mode_ Correct,
         "StSwapped2" %= mode_ Swapped2,
         "StDropLast" %= mode_ DropLast ]

   numberMode = "NrMode" <==
       [ "NrCorrect" %= mode_ NCorrect,
         "NrEmpty" %= mode_ NEmpty ]

   order = "ArgsOrder" <==
       [ "CommonOrder" %= mode_ CommonO,
         "CoCiSN" %= mode_ CoCiSN,
         "NSCiCo" %= mode_ NSCiCo,
         "CoNSCi" %= mode_ CoNSCi ]
```

Figure 2. The classification tree for Google Geocoding API

The tree is specified with respect to a specific address *a*. Given such an address, the tree induces in total 384 test-cases. This is too many for the manual approval stage. On the other hand, the minimal suite will just contain 4 test-case, which does not feel to be adequate. So, we use a rule to select the combinations that we think more important to cover. It is the following rule:

```
r1 = incl ["StCorrect"]
   &&* incl "ArgsOrder"
   &&* incl "CityMode" |+| neg(incl["StCorrect"])
   &&- incl "ArgsOrder" &&-incl "CityMode"
```

Below we show how we construct the final test-suite. Basically it turns the rule above to a test-suite, then we apply minimalistic padding. Then we do few other format-related things, e.g. fixing the order of the classes in each test-case.

```
ggSuite a = ct a $$ rule r1
             >>=  padding
             >>=  fixorder
             >>=  lift_ . strip
```

E.g. this suite will fully combine an un-mangled street-name with the four permutations included in the classification tree and with all mangled variations of the city-name. The resulting combinations are then minmally combined with the other parameters.

The above suite gives 20 test-cases. It is then given to a concretization function to generate the corresponding concrete test-suite.

VII. Related work

Other tools implementing CTM are EXTRACT [9], ADDICT [1], and CTE-XL. The last one, by Lehmann and Wegener [4], is probably the most prominent one. CTE-XL also provides a set of operators to declaratively select a suite. The selection is expressed by two kinds of rules: 'combination rules' and 'dependency rules'. The first are used to specify which base-suite to generate, the second specify constraints on the suite and are used to filter the base-suite. Just as in CTy, any suite can be generated with CTE-XL. However the used operators are different. CTE-XL does not have the `neg`, |+|, and |&| in its combination rules. Having these operators allow us to provide the derived Boolean-like operators $\mathtt{imp}_f$, $\mathtt{imp}_m$, $\mathtt{xor}_f$, $\mathtt{xor}_m$, etc. We can also define the k-wise combination operator as a derived operator. CTE-XL provides Boolean operators to express dependency rules. CTy does not distinguish between the two kinds of rules. It uses the suite-level operators `suchthat` and `except` to express constraints, where any rule or suite can be used as a constraint.

Yu et al [9] proposed an extension to CTM where classes can be annotated with tags (also called 'properties'). Selector expressions are then written to specify that e.g. a class c can only occur in a test case t if t contains (or does not contain) a certain set of tags. The expressiveness of tags is equal to CTE-XL's dependency rules. A class can also be annotated with weight reflecting its importance. The generated test-cases are then sorted so that 'heavy' test-cases will be exercised first. We did not show this here, but in CTy the ability to sort and prioritize test-cases is just a natural extension of suite expressions —see [6].

Singh et al proposed to associate formal predicates expressed in Z to the partitions in a classification tree [7]. This allows us to delay further decomposition of a partition, or to keep it implicit altogether. A test-case *t* induces a formula *f* that logically identifies it, which is just the conjunction of all associated formulas of *t*'s classes (or partitions of lowest level). Modulo decidability, non-sensical test-cases can be filtered out by checking if the conjunction of *f* and SUT's precondition is still satisfiable. By transforming *f* to DNF we can split *t* to a set of more refined test-cases, one for each DNF clause. We did not explore into this direction.

VIII. Conclusion

CTy is an implementation of the CTM approach. However, it is not graphical; it is less suitable for testers who are less proficient in programming. It is provided as a DSL embedded in Haskell. This gives us first class access to Haskell's features. E.g. we get type checking, higher order function, lazy evaluation, and libraries for free. The embedding approach gives us much expressiveness and flexibility, while the syntax is still quite clean. Testers do have to be familiar with some degree of Haskell concepts and syntax.

References

[1] A. Cain, T.Y. Chen, D. Grant, P.L. Poon, S.F. Tang, and T.H. Tse. "An automatic test data generation system based on the integrated classification-tree methodology". In Software Engineering Research and Applications, volume 3026 of LNCS, pages 225–238. Springer, 2004.

[2] M. Grochtmann and K. Grimm. "Classification trees for partition testing". Software Testing, Verification & Reliability, 3(2), pages 63–82, 1993.

[3] S.L.P. Jones. "Composing contracts: An adventure in financial engineering". In Int. Symp. of Formal Methods Europe (FME), volume 2021 of LNCS, page 435. Springer, 2001.

[4] E. Lehmann and J. Wegener. "Test case design by means of the CTE XL". In Proc. of 8th European Int. Conf. on Software Testing, Analysis & Review (EuroSTAR), 2000.

[5] J. Peterson, P. Hudak, and C. Elliott. "Lambda in motion: Controlling robots with Haskell". Volume 1551 of LNCS, pages 91–105, 1999.

[6] I.S.W.B. Prasetya, J. Amorim, T.E.J. Vos, and A. Baars. "CTy: a Haskell DSL for specifying and generating combinatoric test-cases". Technical Report Nr. UU-CS-2011-005, Dept. Inf. & Comp. Sciences, Utrecht Univ., 2011.

[7] H. Singh, M. Conrad, and S. Sadeghipour. "Test case design based on Z and the classification-tree method". In IEEE Int. Conf. on Formal Engineering Methods (ICFEM), pages 81–90, 1997.

[8] S.D. Swierstra. "Combinator parsers --from toys to tools". Electr. Notes Theor. Comput. Sci, 41(1), 2000.

[9] Y.Y. Yu, S.P. Ng, and E.Y.K. Chan. "Generating, selecting and prioritizing test cases from specifications with tool support". In Int. Conf. on Quality Software (QSIC), pages 83–90, 2003.