Working Paper Series
ISSN 1177-777X

Improving our Fitnesse: From Concrete Executions to Partial Specification

David Streader, Mark Utting, Rick Mugridge

Working Paper: 03/2011
April 7, 2011

Improving our Fitnesse: From Concrete Executions to Partial Specification

David Streader, Mark Utting,Rick Mugridge
Department of Computer Science,
University of Waikato, Hamilton, New Zealand
{dstr,marku}@cs.waikato.ac.nz

April 7, 2011

Abstract

Fitnesse and FIT [5] allow systems tests to be written by non-programmers using a Wiki or HTML style of input. However, there is little support for syntactic and semantic checks as the tests are being designed. This paper describes a support tool for designing table-based test cases that gives deep semantic analysis about a set of test cases. It uses a variety of strategies such as pairwise analysis, boundary value analysis and test case subsumption to suggest missing test cases and to generalise concrete tests into more abstract tests. The goal is to interactively improve the quality of test suites during the test design phase.

1 Introduction

From requirements capture to test design, people think about the behavior of a system both concretely (one test at a time) and abstractly (over an entire set of tests). Tool support tends to be better at supporting the concrete view (a single test), rather than analyzing the interrelationships and overall coverage of a set of tests. Yet gaining feed back from sets of requirements/tests could help ensure their accuracy and improve their quality. What we describe here is a system that provides semantic feed back that may help in the construction of the requirements of a system or the construction of tests for a system. It is intended to be one component of an interactive development environment for designing Fitnesse/FIT ColumnFixture tests [5], which are HTML tables containing several rows, with each row representing one independent test.

The designer inputs the details of concrete requirements/tests into an Input table, one row per requirement/test, and in the background the system analyzes the input data and offers suggestions in a separate Analysis table. The designer is able to *promote* a suggestion from one row of the Analysis table to the Input table when ever they wish to.

We have explored several algorithms for generating suggestions. Several of the algorithms suggest additional test cases that seem to be missing from the input table.

We also provide facilities for pointing out redundant input cases or cases that contradict each other. Other algorithms generate suggestions that try to *abstract* one or more of the existing tests in the input table, in order to generalise some of the input table.

Section 2 introduces some example requirements, Section 3 briefly describes the test design tool, Section 4 discusses algorithms for suggesting missing tests/requirements, Section 5 discusses algorithms for abstracting concrete tests to give a higher-level view of the tests, and Section 6 discusses our conclusions.

2 Example Selling Widgets for a profit

In this section we consider a simple example where the natural language description of the system (tests) is given in Section 2.1. From this a table of concrete requirements/tests as described in Section 4.3 can be constructed.

2.1 Requirements and/or tests for Selling Widgets

The cost of our high tech Widgets is dominated by the research and development costs, their manufacturing costs is relatively insignificant. The end-user cost of our Widgets (including shipping) is specified informally as follows:

We sell widgets both locally and to two foreign countries. For local customers we sell 10 *or less widgets for* $\$120$ *and thereafter* $\$8$ *for each additional widget until the cost is* $\$4000$ *and we can send a van load. The van holds up to* 550 *widgets and we make no extra charge for the widgets that fill the van. After filling one van we charge* $\$5$ *for each additional widget. For customers in country* India *that wish the widgets to be air freighted the cost is* $\$20$ *per widget. If customers chose sea freight then the cost is* $\$10000$ *per container of* 1000 *widgets, or part thereof. For customers in country* China *that wish the widgets to be air freighted the cost is* $\$16$ *per widget.*

2.2 Gathering requirements/tests

We start with an unstructured collection of known concrete cases. What we need is greater confidence in both the specification and the set of cases. In particular we would like reassurance that the specification is consistent and, to some degree, complete. Whether or not an enumerated set of cases completely cover all relevant points of an informal specification can often only be decided by the human analyst.

3 Brief description of tool

This tool is designed to help us understand what might be useful when specifying an operation via concrete examples of input and output.

Users input an Input Table containing several rows, where each row contains several concrete input values for an operation and, in the final column, the output value for the operation. The tool builds an Analysis Table consisting of rows that are consistent with the input table. The user can add one of the suggested rows to their input table.

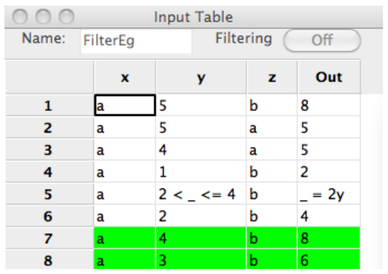
Country	Num	Shipping	Cost
local	1	Air	\$100
local	11	Sea	\$128
local	495	Air	\$4000
local	550	Sea	\$4000
local	600	Air	\$4400
local	650	Air	\$4800
India	1	Air	\$20
India	2000	Sea	\$20000
India	550	Air	\$11000
India	1000	Air	\$20000
China	1	Air	\$16

Figure 1: Tabulated Specification

The input table is checked for self consistency and any line inconsistent with any earlier line (appearing above in the table) is flagged as inconsistent and displayed in red.

Some suggestions may contain abstract values, such as 'Any' (any value of the appropriate value in this column) or '$\_ < 2$' (any value less than 2). These abstract rows can be added to the **Input Table** if the user is confident that they correctly model the desired behavior. Adding one row of the analysis table into the input table always preserves the consistency of the input table, but adding multiple rows simultaneously could break the consistency, because each suggestion is independent.

If one row is more abstract than another we say that the more concrete row can be *inferred* from the more abstract. When the table is displayed, any row that can be inferred from another row is shaded (green) and placed at the bottom of the table see Fig 2.

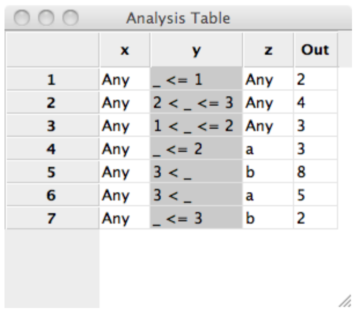


Name: FilterEg Filtering Off

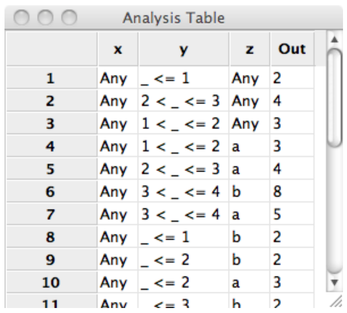
	x	y	z	Out
1	a	5	b	8
2	a	5	a	5
3	a	4	a	5
4	a	1	b	2
5	a	2 < _ <= 4	b	_ = 2y
6	a	2	b	4
7	a	4	b	8
8	a	3	b	6

Figure 2: Shaded rows infered from earlier rows

The tool also has an option for automatically filtering out all rows in the suggestion table that are more concrete than another suggestion.
In Fig 3 the left hand image is a filtered output of seven rows where as the first 10 rows of the unfiltered output appear on the right. The first three rows of the unfiltered


Analysis Table

	x	y	z	Out
1	Any	_ <= 1	Any	2
2	Any	2 < _ <= 3	Any	4
3	Any	1 < _ <= 2	Any	3
4	Any	_ <= 2	a	3
5	Any	3 < _	b	8
6	Any	3 < _	a	5
7	Any	_ <= 3	b	2


Analysis Table

	x	y	z	Out
1	Any	_ <= 1	Any	2
2	Any	2 < _ <= 3	Any	4
3	Any	1 < _ <= 2	Any	3
4	Any	1 < _ <= 2	a	3
5	Any	2 < _ <= 3	a	4
6	Any	3 < _ <= 4	b	8
7	Any	3 < _ <= 4	a	5
8	Any	_ <= 1	b	2
9	Any	_ <= 2	b	2
10	Any	_ <= 2	a	3
11	Any	_ <= 3	b	2

Figure 3: Filtered and unfiltered Analysis

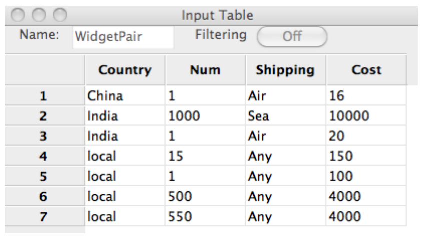
analysis table apper in the filtered table but the forth row dose not. This is because it can be inferred from the third row.

4 Suggesting Missing Requirements/Tests

In this section we describe several algorithms we have used to suggest possible concrete test cases that may be missing or worth adding to the input table. These are based on common test design heuristics, such as pairwise coverage of inputs [1] and Modified Condition/Decision Coverage analysis

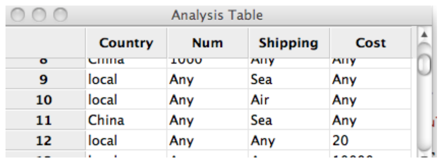
4.1 Pairwise

The Pairwise algorithm builds the domain of input values for each input column then in turn takes pairs of domains and builds the cartesian product of these domains. Then for any pair of inputs that is not in the original table it suggests a row containing that pair with all other input columns set to Any and the output left blank.


Input Table

Name: WidgetPair   Filtering: Off

	Country	Num	Shipping	Cost
1	China	1	Air	16
2	India	1000	Sea	10000
3	India	1	Air	20
4	local	15	Any	150
5	local	1	Any	100
6	local	500	Any	4000
7	local	550	Any	4000


Analysis Table

	Country	Num	Shipping	Cost
8	China	1000	Any	Any
9	local	Any	Sea	Any
10	local	Any	Air	Any
11	China	Any	Sea	Any
12	local	Any	Any	20

Figure 4: Pairwise suggestions

The result of promoting more than one suggestion from these Pairwise Analysis Tables can result in an inconsistent Input Table.

In the example in Fig 4 row 11 (of 63) in the Analysis Table is the pair China and Sea. This is saying that nothing has been specified for this case, which may indicate an ommision in the specification in Section 2.1. To complete the informal specification we add:

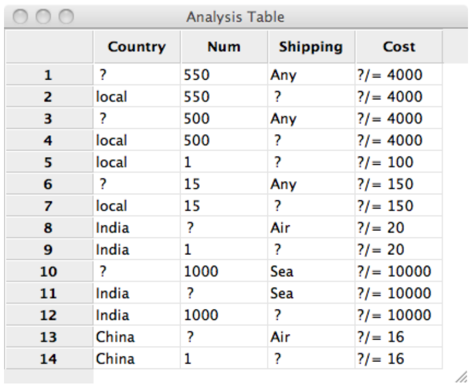
If customers in in China choose sea freight then the cost is $8000 *per container of* 1000 *widgets, or part thereof.*

Although these suggestions can prompt the analyst to consider new and potentially interesting cases, they may be lost in a blizzard of uninteresting suggestions. How best to filter or order the suggestions from the pairwise analysis is an open question and may depend upon the user.

4.2 Modified Condition/Decision Coverage

Modified Condition/Decision Coverage (MC/DC), has been used to ensure that software is tested adequately [2] and has been used for test suit reduction [4]. In our simple requirement capture/test suit definition situation we apply this principle to give feed back about potential gaps in the coverage of the requirements/tests.

The output from applying MC/DC analysis to the input table in Fig 4 can be see in Fig 5.


Analysis Table

	Country	Num	Shipping	Cost
1	?	550	Any	?/= 4000
2	local	550	?	?/= 4000
3	?	500	Any	?/= 4000
4	local	500	?	?/= 4000
5	local	1	?	?/= 100
6	?	15	Any	?/= 150
7	local	15	?	?/= 150
8	India	?	Air	?/= 20
9	India	1	?	?/= 20
10	?	1000	Sea	?/= 10000
11	India	?	Sea	?/= 10000
12	India	1000	?	?/= 10000
13	China	?	Air	?/= 16
14	China	1	?	?/= 16

Figure 5: MC/DC suggestions

The MC/DC output contains only 14 rows which compares favorably to the pairwise output of 63 rows. Row 10 in Fig 5 tells us that data for shipping 1000 Widgets by sea appears for only one country .

4.3 Concrete requirements/tests

After considerable effort we construct a set of requirements/test castes that we believe to be consistent and, in some sense, cover the important points.

Country	Num	Shipping	Cost
local	1	Air	$100
local	11	Air	$108
local	495	Air	$4000
local	550	Air	$4000
local	600	Air	$4250
local	650	Air	$4500
India	1	Air	$20
India	11	Air	$220
India	550	Air	$1100
India	1000	Air	$20000
China	1	Air	$16
China	11	Air	$176
China	550	Air	$880
China	1000	Air	$16000

Country	Num	Shipping	Cost
local	1	Sea	$100
local	11	Sea	$108
local	495	Sea	$4000
local	550	Sea	$4000
local	600	Sea	$4250
local	650	Sea	$4500
India	1	Sea	$10000
India	1000	Sea	$10000
India	2000	Sea	$20000
India	3000	Sea	$30000
India	500	Sea	$10000
India	2100	Sea	$20000
China	1	Sea	$8000
China	1000	Sea	$8000
China	2000	Sea	$16000
China	3000	Sea	$24000
China	500	Sea	$8000
China	2100	Sea	$16000

5 Analysis by abstraction

By allowing rows to contain non-concrete values such as Any or inequalities or ranges of values, we can introduce an *abstraction* relationship between rows. Considering input columns only, we define the semantics of a row to be the set of *concrete* test cases that satisfy the requirements of the row. So the semantics of a concrete test case is the singleton set containing that test case (just its input values). We define the abstraction relation between rows to be the subset relation, which gives us a lattice of row abstractions (though only some points in the lattice will be expressible using the limited syntax that we allow in input rows). The top of the lattice is the row that has Any in every column, the bottom of the lattice is the empty set, which represents an inconsistent row, and all concrete test cases are positioned immediately above this bottom element, since they are singleton sets. The Hasse diagram [3] in Fig 6 shows how several example rows fit into the abstraction lattice.

The algorithms in this section make use of this abstraction lattice in various ways.

5.1 Independence

This algorithm views all inputs as being taken from an enumerated set. It takes an input row and generalizes it by replacing one of the concrete values by Any. However, it only does this if the resulting abstract test is consistent with every other row in the current input table see Fig 7. The rationale for doing this is that the behaviour of the underlying system under test, SUT, may be *independent* of that concrete value, because it is determined by other values in that row.

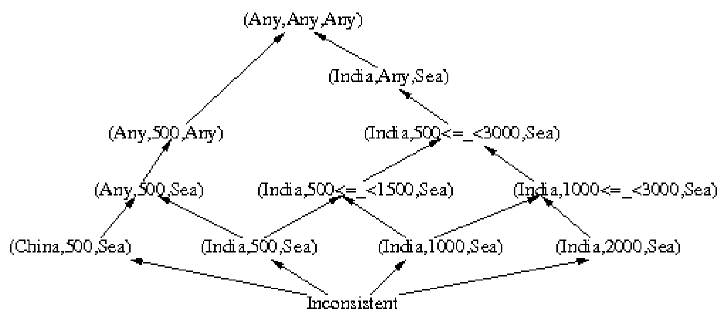
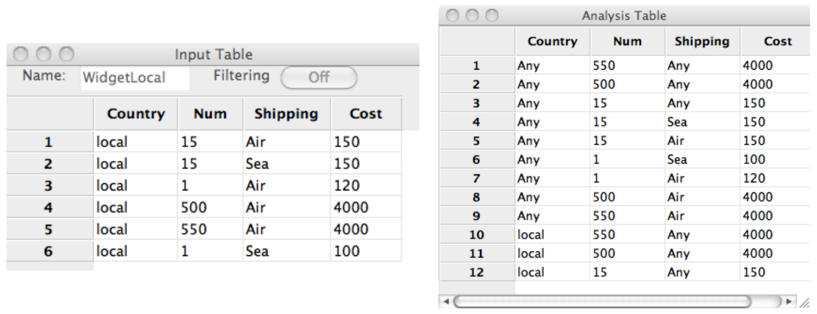


Figure 6: Hasse diagram of the abstraction lattice, showing the abstraction relation between four example rows.



Input Table

Name: WidgetLocal Filtering Off

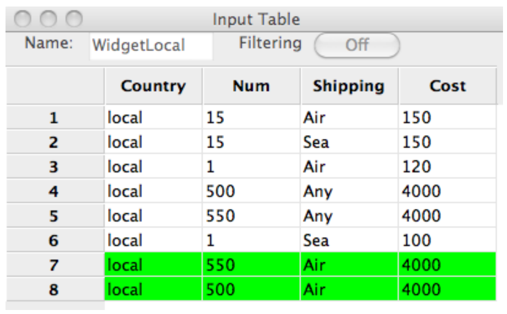
	Country	Num	Shipping	Cost
1	local	15	Air	150
2	local	15	Sea	150
3	local	1	Air	120
4	local	500	Air	4000
5	local	550	Air	4000
6	local	1	Sea	100

Analysis Table

	Country	Num	Shipping	Cost
1	Any	550	Any	4000
2	Any	500	Any	4000
3	Any	15	Any	150
4	Any	15	Sea	150
5	Any	15	Air	150
6	Any	1	Sea	100
7	Any	1	Air	120
8	Any	500	Air	4000
9	Any	550	Air	4000
10	local	550	Any	4000
11	local	500	Any	4000
12	local	15	Any	150

Figure 7: Independence Anylisis

Adding any one of these rows to the input table results in a consistent table. In our running example we know that the Cost for local delivery is indelendent on the Shipping type. Hence we can promote lines ten and eleven from the Analysis Table. This increased the coverage of the Input Table, shown in Fig 8 gives us increased confidence that we have not omitted important detail.



Input Table

Name: WidgetLocal Filtering Off

	Country	Num	Shipping	Cost
1	local	15	Air	150
2	local	15	Sea	150
3	local	1	Air	120
4	local	500	Any	4000
5	local	550	Any	4000
6	local	1	Sea	100
7	local	550	Air	4000
8	local	500	Air	4000

Figure 8: Promoted abstract rows

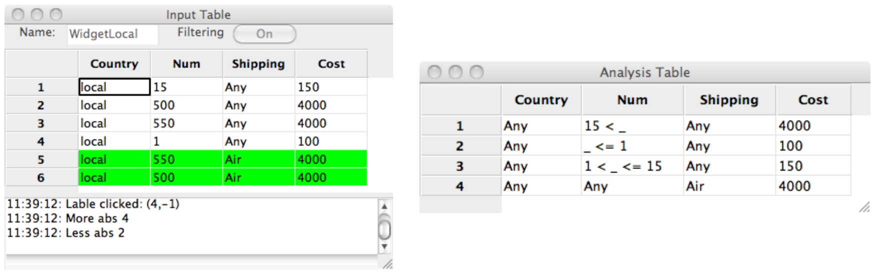
But viewing the Input Table in Fig 8 we can see the need to make the first row more abstract and that this is not on offer in the Analysis Table. This is because we have made a mistake in either rows 1 or 4. This is easy to correct and for this simple example may well have been spotted by an observant reader. This illustrates that the construction of rows for both concrete examples and more abstract specifications can catch mistakes early in the requirements capture or test design.

5.2 Boundary analysis

Boundary value analysis is a software testing heuristic that says that tests should include values on each side of a boundary where behaviour changes. This can be particularly useful as these boundaries are common locations for software faults and are frequently used as test cases. The boundary points can be provided explicitly, by the analyst defining the output values on either side of the boundary, or can be guessed at by the tool assuming some interpolation rule. For example in Section 5.3 we assume that the output value changes in a step wise fashion as an ordered input changes and in Section Section 5.4 we assume a linear relationship between one of the input columns and the output column.

5.3 Range Simplification

This algorithm first splits any ordered column into ranges cut by any number of values input in the column. Then, for each column, it builds a subset ordering between the ranges. For enumerated types like strings it just takes the domain and add the wild card Any but for columns of ordered elements like Nat or $Rational$ it builds the set of all intervals from the domains. Finally, we build the Cartesian product of all ordered domains keeping only those that are consistent with all rows in the Input Table.



	Country	Num	Shipping	Cost
1	local	15	Any	150
2	local	500	Any	4000
3	local	550	Any	4000
4	local	1	Any	100
5	local	550	Air	4000
6	local	500	Air	4000

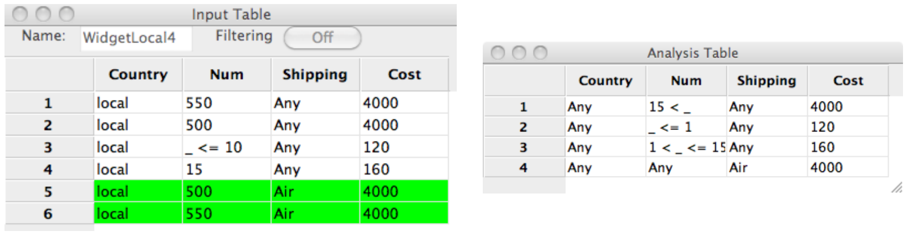
```
11:39:12: Lable clicked: (4,-1)
11:39:12: More abs 4
11:39:12: Less abs 2
```

	Country	Num	Shipping	Cost
1	Any	15 < _	Any	4000
2	Any	_ <= 1	Any	100
3	Any	1 < _ <= 15	Any	150
4	Any	Any	Air	4000

Figure 9: Range simplification

We have applied range simplification to the new Input Table for our example and to reduce the clutter in the resulting Analysis Table we have switched the filtering on. But when we look at the results in Fig 9 we can see that the cost of one widget is not the same as for two widgets and we again must have made a mistake on input. Returning to the text specification we see that the cost of 1 to 10 widgets is $120 not

\$100. Making these two corrections and applying the range simplification results in the Analysis Table on the left of Fig 10.



Input Table

Name: WidgetLocal4 Filtering Off

	Country	Num	Shipping	Cost
1	local	550	Any	4000
2	local	500	Any	4000
3	local	_ <= 10	Any	120
4	local	15	Any	160
5	local	500	Air	4000
6	local	550	Air	4000

Analysis Table

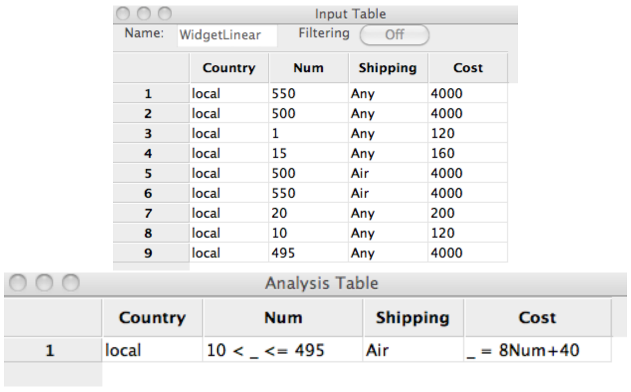
	Country	Num	Shipping	Cost
1	Any	15 < _	Any	4000
2	Any	_ <= 1	Any	120
3	Any	1 < _ <= 15	Any	160
4	Any	Any	Air	4000

Figure 10: Corrected input

Alas this is still not quite what we want, as the tool infers a boundary point of 1, rather than the correct boundary of 10. This has exposed another inadequacy in our set of input tests, which can can easily be corrected, either by adding a concrete row to the Input table for the cost of sending 10 widgets, and then reapplying the range simplification, or by promoting row 3 and editing it in the Input Table on the right of Fig 10.

5.4 Linear Simplification

When the output Cost and one input Num are from an ordered domain like Nat or Rational then they might be related via a simple linear equation $\text{Cost} = a * \text{Num} + b$. The Linear Simplification algorithm searches for three or more concrete instances in a row that fit a linear equation. When it finds them it constructs an analysis row containing the equation in the output Cost column and a range in the variable input column Num, as shown in Fig 11.



Input Table

Name: WidgetLinear Filtering Off

	Country	Num	Shipping	Cost
1	local	550	Any	4000
2	local	500	Any	4000
3	local	1	Any	120
4	local	15	Any	160
5	local	500	Air	4000
6	local	550	Air	4000
7	local	20	Any	200
8	local	10	Any	120
9	local	495	Any	4000

Analysis Table

	Country	Num	Shipping	Cost
1	local	10 < _ <= 495	Air	_ = 8Num+40

Figure 11: Linear simplification

5.5 Abstract requirements/tests

If we combined all the above improvements and abstractions into one single table, we would obtain the following table.

Country	Num	Shipping	Cost
India	Any	Air	$Num \times \$20$
India	$1 <= 1000$	Sea	\$10000
India	$1001 < \_ <= 2000$	Sea	\$20000
India	$2001 < \_ <= 3000$	Sea	\$30000
China	Any	Air	$Num \times \$16$
China	$1 <= 1000$	Sea	\$8000
China	$1001 < \_ <= 2000$	Sea	\$16000
China	$2001 < \_ <= 3000$	Sea	\$24000
local	$0 < \_ <= 10$	Any	\$120
local	$10 < \_ <= 495$	Any	$\$40 + (Num * 8)$
local	$495 < \_ <= 550$	Any	\$4000
local	$550 < \_$	Any	$\$4000 + (Num - 500) \times 5$

The abstract requirements are much shorter than the concrete input table. This makes it easier to see what has been covered and that there are no inconsistencies.

6 Conclusions

We have described a semantic analysis tool for Fitnesse ColumnFixture tables, or any kind of test input that consists of multiple tuples of concrete input values. The tool can detect inconsistencies in the table and suggest missing tests, based on various common test design heuristics.

By extending the output notation (the Analysis Table) from considering only literal inputs, to allowing more abstract inputs such as arbitrary values Any and ranges of values $\_ <= 2$, we can generate more abstract views of the input tests. That is, it becomes possible to specify the output of a method/function over a potentially infinite domain with only a small handful of statements. As we have shown, this allows a single requirement/test to be defined in different ways and hence has the potential to highlight errors early in the requirements/test design process.

6.1 Interesting Extensions

Improved filtering would be possible by making use of the *type* of each column. Extensions to what terms can be parsed could greatly expand what can be represented by a single input row.

Analysing the content of multiple tables at once would also greatly increase the expressive power of small sets of rows. For example a container is never empty and contains 1000 or less widgets and, in addition, a new container is never started until existing containers are full.

Container	Num
Any	$1 + 1000(Container - 1) <= \_ <= 1000Container$

Using this fact, Table, we can define a simple linear relation between the number of containers and the cost of shipping by sea.

Country	Shipping	Container	Cost
India	Sea	Any	$Container \times \$10000$
China	Sea	Any	$Container \times \$8000$

This results in a simpler specification with improved coverage.

References

[1] Jacek Czerwonka. Pairwise testing web site. `http://pairwise.org`, March 2011.

[2] A. Dupuy and N. Leveson. An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In *Digital Avionics Systems Conferences, 2000. Proceedings. DASC. The 19th*, volume 1, pages 1B6/1 –1B6/7 vol.1, 2000.

[3] Ralph Freese. Automated lattice drawing. In *Concept Lattices*, number 2961 in Lecture Notes in Computer Science, pages 589–590. Springer-Verlag, 2004.

[4] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29:195–209, 2003.

[5] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, 2005.