Migrating combinatorial interaction test modeling and generation to the web

Angelo Gargantini
DIGIP University of Bergamo
Bergamo, Italy
angelo.gargantini@unibg.it

Marco Radavelli
DIGIP University of Bergamo
Bergamo, Italy
marco.radavelli@unibg.it

***Abstract*—Combinatorial Interaction Testing (CIT) is an effective technique that, however, requires a good tool support in order to be successfully applied. There are several tools and applications for CIT, and most of them are distributed as desktop applications or as special plugins of existing programs and require some installation procedures to be used. Software as a Service (SaaS) paradigm can be applied to CIT modeling and test generation, proving several advantages to the tester. There are already some attempts in this direction, but with some shortcomings (little editing support, for example). In this paper, we present CTWEDGE (Combinatorial Testing Web EDiting and GEneration), which defines a language for CIT models in the presence of constraints by using Xtext. It introduces a web service for editing CIT specifications and it interfaces, through a server, with other tools for test generation. CTWEDGE can improve the user experience by providing a complete environment for CIT modeling and generation on the web without sacrificing usability.**

***Index Terms*—combinatorial interaction testing, Software as a Service, domain specific language, web editor**

I. Introduction

Combinatorial Interaction Testing (CIT) is becoming a widespread practice for software testing. The presence of tools for CIT is of fundamental importance because performing CIT activities manually can be error prone and time consuming. The site pairwise.org[1] lists at least 43 tools supporting several CIT activities while the paper [1] reviewed the CIT literature and found around 20 tools. Most of the tools are classical programs or plugins of existing programs/platforms. In all these cases, the user has to download, install, and execute the program on his/her machine. Some tools offer a GUI interface, for example, for defining the models and run the test generation, while others rely on other programs for some activities (like PICTMaster, based on PICT [2], that works entirely inside Microsoft Excel). In [3] we have presented a plugin for the eclipse IDE that helps the user in writing CIT models with constraints by leveraging all the expected features of a modern IDE, and it generates combinatorial test suites by using third party programs (like CASA or ACTS).

However, this classical approach poses some challenges. First, the user must install the chosen CIT tool, which in turn may require some dependencies, like java, or in case of plugin, it requires the installation of another program or application like eclipse, excel and so on. Then the user must use his/her machine for running the test generation algorithms. For experienced users with powerful machines they can administer, this is not a real problem. However, for novice users with not so powerful computer, or students that are just learning the CIT principles, or software developers using computer they cannot administer, this can become a cost to be considered and it may be an obstacle for the use of CIT.

Second, from the point of view of tool developers, the distribution of programs means that they have little or no control on the software once is installed: when a bug is discovered the developer has to fix the bug, publish a new version of the tool and hope that the users will update their software. Moreover, the developer has no idea about how the software is used: what are the typical scenarios of use (big or small models, for example), what are the features that are mostly used (for example, what modeling features are more used). Furthermore, it is difficult for tool developers to apply a cost model able to reward their effort in developing and deploying the CIT tool. Indeed, most of the tools are given away for free by researchers supported by their own organization.

A possible solution of these problems could be the offer of CIT features as software as a service (SaaS). SaaS is software that is accessed through Internet by using a classical web browser. The SaaS software is actually hosted on the vendors servers, and the customers log in and perform tasks as necessary. The vendor is the one who is ultimately responsible for hosting, upgrading and maintaining the program as needed. There is an extensive literature about SaaS and the advantages (together with limitations) are well known [4].

There are already experiences in using the web for CIT. For instance, CTWeb Classic [5], CTWeb Plus [6], TestCover [7], Hexawise [8], and PairWiser [9].

However, as we will discuss in the related work section, each tool has its own specific language to define combinatorial models, with its own predefined output formats and a (limited) set of supported existing or in-house algorithms for test case generation. Not all those tools are equally powerful or easy to use, and many of them are commercial or require registration.

For this reason, starting from our experience with CitLab [3], we have worked on a web based application that allows the user to write CIT models in a similar way he/she

[1]See http://www.pairwise.org/tools.asp

```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines: [ 25 .. 30 ]
   display : {16MC, 8MC, BW}
```

Fig. 1: A smartphone example

would do in a classical IDE and it offers test suite generation by means of server using known and community evaluated test generation algorithms. Our system, called CTWEDGE (Combinatorial Testing Web EDiting and GEneration), introduces a rich language for combinatorial models, offers a powerful web editor, and allows the user to generate the CIT test suites on a server. The only software needed to use CTWEDGE is a modern web browser.

The paper is organized as follows. In Sect. II we present the modeling language (in an abstract way) we use to define CIT models. In Sect. III we introduce our tool, its architecture, its web editing capabilities, and the generator engine. A detailed comparison with other similar web based tools is presented in Sect. IV. Future works and possible directions of extension are presented in Sect. V. Sect. VI concludes the paper.

II. A SIMPLE LANGUAGE FOR CIT MODELS

We have devised a simple textual language for CIT models which is suitable to be used in web editors. It allows the definition of parameters, each with its name and (finite) domain, and it is based on our previous language defined for CITLAB [3]. We allow the following parameter types:

1) Boolean with only two possible values true and false (all lower or all upper case). The two boolean constants are also considered of Boolean type.
2) Ranges that are integer intervals defined by their lower bound l and upper bound u. With $[l..u]$ we denote all the integers between l and u included.
3) Enumerative that are a list of possible values between {}. We are rather liberal about the elements and we allow identifier starting also with a number, natural numbers, and strings. For example, one could define a enumerative values in this way:

```
values : {100, 1M, "my␣name", cit}
```

A simple example of combinatorial model with three parameters is shown in Fig. 1.

There are some semantic rules about the parameters and their definitions, defined as follows:

1) The name of each parameter must be unique.
2) In Ranges, the lower bound l must be less than the upper bound u.
3) The elements in each Enumerative must be distinct. We allow two enumerative parameters to share some elements, though.

A. Constraints

A distinctive feature of our langauge is the support of modeling constraints among parameters. In most configurable systems, constraints or dependencies exist between parameters. Constraints may be introduced for several reasons, for example, to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply design choices [10]. In our approach, tests that do not satisfy the constraints are considered *invalid* and do not need to be produced. For this reason, the presence of constraints may reduce the number of tests of the final test suite (but it may also increase it [10]). However, the generation of tests considering constraints is generally more challenging than the generation without them, and several test generation techniques still do not support constraints, at least not in a direct manner. In CTWEDGE we decided to focus more on techniques supporting constraints.

In CTWEDGE, we adopt the language of propositional logic (with the usual logical operators) with equality and arithmetic to express constraints. To be more precise, we use propositional calculus, enriched by the arithmetic over the integers and enumerative symbols. As operators, we admit the use of equality and inequality for any variable, the usual Boolean operators for Boolean terms, and the relational and arithmetic operators for numeric terms. To be more precise, Table I reports all the rules we have defined to check if a constraint is semantically correct.

For example, we can write constraints in this way:

```
Model Phone
..
Constraints:
# emailViewer => textLines > 28 #
# emailViewer and display != 16MC => textLines > 28 + 3#
```

Many test suite generation tools provide a limited support for constraints. For instance, AETG [11], [12] allows only simple constraints of type if then else or requires. The language of CTWEDGE is in this aspect more expressive, as it is targeted to be more general than existing tools. In the specific case of AETG, the translation of those templates into our logic is straightforward. For example the if then else constraint can be translated by two implications. Other tools [10] allow constraints only in the form of forbidden combinations [13]. Our language is more general, as a forbidden tuple would be translated as a *not* statement. For instance, a forbidden pair emailViewer = false; display = 16MC would be represented by the following constraint:

```
# not (emailViewer = false and display = 16MC) #
```

However, the explicit list of the forbidden combinations may explode and it may become impractical and error-prone to represent it. For example, if the model of mobile phones presented in Fig. 1 had a constraint that *"A front video camera requires also a 16MC display"*. This constraint would be translated into two forbidden tuples:

```
(emailViewer = true, display = 8MC);
```

Expression	Case	Correct iff
e_1 *op* e_2 with *op* $\in \{=, \neq\} \rightarrow$ *boolean* e_2 *op* e_1 with *op* $\in \{=, \neq\} \rightarrow$ *boolean*	e_1 enumerative, e_2 element	e_2 belongs to e_1 elements
	e_1 enumerative, e_2 enumerative	e_1 and e_2 share at least one element
	e_1 range, e_2 range	e_1 and e_2 share at least one number
	e_1 range, e_2 number	always
	e_1 number, e_2 number	always
	e_1 boolean, e_2 boolean	always
e_1 *op* e_2 with *op* $\in \{<, \leq, >, \geq\} \rightarrow$ *boolean* e_2 *op* e_1 with *op* $\in \{<, \leq, >, \geq\} \rightarrow$ *boolean*	e_1 range, e_2 range	e_1 and e_2 share at least one number
	e_1 range, e_2 number	always
	e_1 number, e_2 number	always
e_1 *op* e_2 with *op* $\in \{\wedge, \vee, \rightarrow\} \rightarrow$ *boolean* e_2 *op* e_1 with *op* $\in \{\wedge, \vee, \rightarrow\} \rightarrow$ *boolean*	e_1 boolean, e_2 boolean	always
$\neg e \rightarrow$ *boolean*	e boolean	always
e_1 *op* e_2 with *op* $\in \{+, -, *, /, \%\} \rightarrow$ *number*	e_1 number, e_2 number	$e_2 \neq 0$ if $op = /$ or $op = \%$
	e_1 range, e_2 range	always
	e_1 range, e_2 number	$e_2 \neq 0$ if $op = /$ or $op = \%$
	e_1 number, e_2 range	always

TABLE I: Rules of CTWEDGE Language Validator for Constraints

```
(emailViewer = true, display = BW);
```

However, the translation as constraint in general form would be simply:

```
# emailViewer => display = 16MC #
```

which is more compact and more similar to the informal requirement.

In our language semantics, a test case is valid only if it does not contradict any constraint in the specification. Others [14] distinguish between combinations to be avoided *if possible* (soft constraints), and the forbidden combinations (hard constraints), which must always be avoided (our case).

Other tools, like CASA [15], support only constraints in conjunctive normal form, without arithmetic or relational operators.

B. Xtext

There are countless ways to define a language together with its parser. One emerging technique for Domain Specific Language modeling is the use of Xtext [16]. By defining the grammar of the DSL of choice by means of a Xtext grammar, the language designer obtains a parser, APIs to programmatically access models, a serializer and a smart editor for it. The editor provides many features out-of-the-box, such as syntax highlighting, content-assist, folding, jump-to-declaration and reverse-reference lookup across multiple files. We had already used Xtext for defining the language in CITLAB [3]. Xtext support also the generation of a web editor, as we present in the following section.

Grammar rules written in Xtext are very close to the standard (E)BNF production rules. For instance, the main grammar rule that defines the whole CIT model is defined as follows:

```
CitModel:
 'Model' name=ID
 'Parameters' ':' (parameters+=Parameter)+
 ('Constraints' ':' (constraints+=Constraint)+)?
```

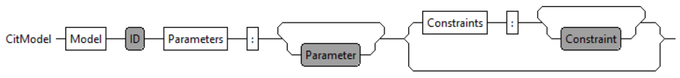


Fig. 2: `CitModel` rule diagram

Xtext can display the production rules by means of syntax diagrams, also known as railroad diagrams. For example, the rule presented before for `CitModel` is shown in Fig. 2.

Because Xtext is based on ANTLR, it does not allow left recursive parser rules and parsing nested expressions is not as simple as writing a EBNF rule. The CTWEDGE language parses the constraints by defining the precedence among operators implicitly by left-factoring expression definitions. For example, in order to parse the AND operators before the OR operators, CTWEDGE introduces the following two rules that are not left recursive:

```
OrExpression returns Expression:
  AndExpression ({OrExpression.left=current}
          OR_OPERATOR (right=AndExpression))*;

AndExpression returns Expression:
  EqualExpression ({AndExpression.left=current}
        AND_OPERATOR (right=EqualExpression))*;
```

The definitions of semantic constraints in Xtext is performed by user-defined validator classes written in Java or in Xtend containing methods annotated by `@Check`. For instance, to check that in the definition of any range domain of our CIT models the upper bound is greater than the lower bound, we have introduced the following checking method:

```
@Check
def checkRangeIsCorrect(Range range) {
 if (range.getBegin() >= range.getEnd())
   error("The␣second␣term␣must␣be␣greater␣...");
}
```

III. CTWEDGE: CT Web Editor and Generator

In this section, we present our tool CTWEDGE. As we can see in Fig. 3, the tool is composed by a language definition component (with its Xtext parser and validator), a web-based

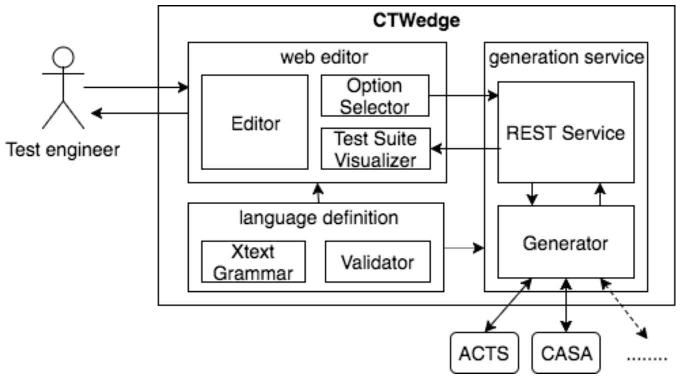


Fig. 3: CTWEDGE architecture

editor for the CTWEDGE language, with some options for test generation and a test suite visualizer and exporter, and a test suite generator that exploits third-party test generation tools.

The tool is written in Java and Xtext, and can be deployed on any Web-application server, such as Apache Tomcat. It is publicly available at: http://foselab.unibg.it/ctwedge/.

A. *Combinatorial Testing Web Editor*

In order to implement a web-based editor, we can leverage the Xtext framework, since Xtext starting from version 2.9 offers an interface for integration of text editors in web applications. The text editors are implemented in JavaScript, and language-related services such as code completion are realized through HTTP requests to a server-side component.

The Xtext web-based editor provides several features, like content assist to help the user to complete the models, validation to check the correctness, syntax coloring, and formatting.

The CTWEDGE web editor is based on Ace (Ajax.org Cloud9 Editor)[2], but other web editors (like Orion and CodeMirror) are available. Xtext does not yet provide support for the recently standardized Language Server Protocol [17], which we plan to include in our tool as a future work. A screenshot of the CTWEDGE web editor is shown in Fig. 4. The web editor provides an immediate feedback while writing by means of syntax highlighting, auto completion, and errors markings.

The validation of the model is performed run-time while the user writes it. If the validator finds an error in the model, it generates an error message. The nature of the error is indicated in the pop-up box appearing when positioning the cursor over the error sign, and the point in which the error occurs is marked in the editor. Fig. 5 shows how model validation errors are displayed to the test engineer.

The editor allows to load predefined examples of combinatorial models, selected from literature [18] and converted into CTWEDGE language format (with extension *.ctw*).

Graphical components (buttons and option selectors) are built using the JavaScript frameworks JQuery[3] and Bootstrap[4].

[2]See https://ace.c9.io/
[3]JQuery: https://jquery.com/
[4]Bootstrap: https://getbootstrap.com/

Parameter	type	description	values
model	String	the combinatorial model	as written and validated by the editor (see Sec. II)
strength	Int	the combinatorial interaction strength	any integer above 1 (default is 2: pairwise)
generator	Enum	the tool to be used	["acts", "casa"] (so far)
ignConstr	Boolean	if constraints should be ignored in test generation	["true", "false"] (default is false)

TABLE II: Request parameters to CTWEDGE generation service

The web application is fully compatible also with mobile devices (Android and iOS), from any recent Web browser with JavaScript enabled.

B. *Test generator web service*

The function of actually generating a test suite from the test model and option parameter is web-served and performed by the test generation service which is the component of CTWEDGE. The test generation service is composed by two modules: a REST[5] service and a language translator.

The REST service handles input model and option parameters for the generation, calls the generators from which it gets back the tests and it is responsible to deliver them to the web browser.

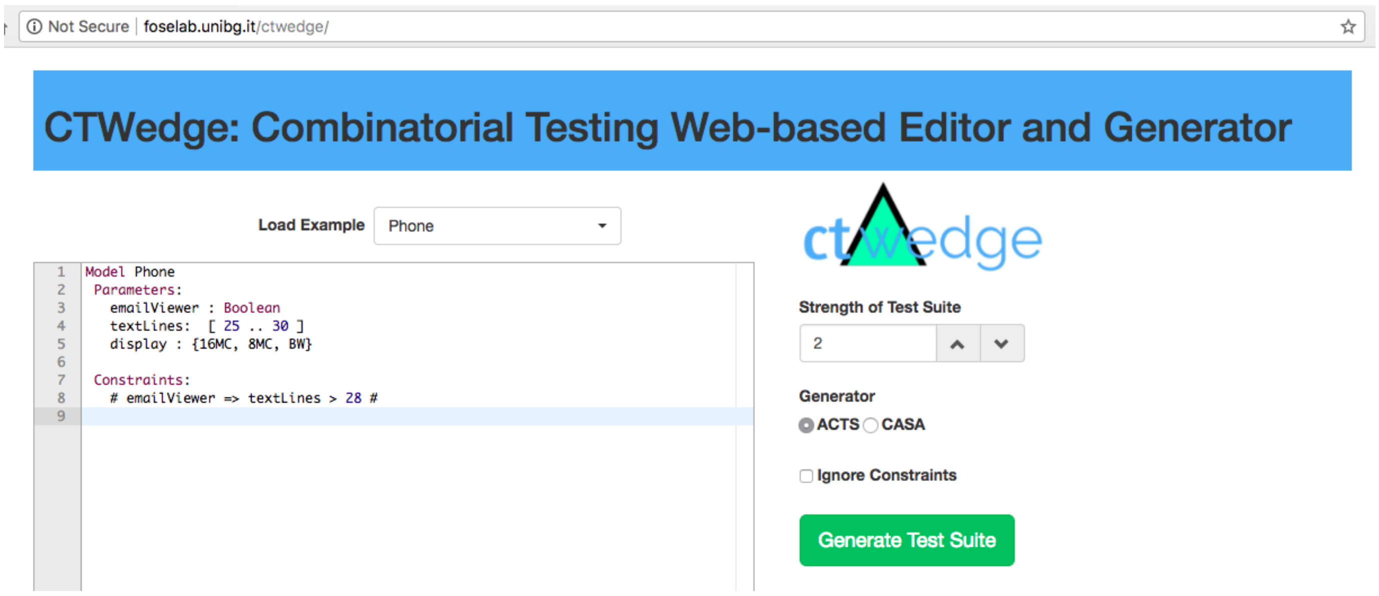
The test generator acts as a *driver* between CTWEDGE and the various third party combinatorial test generation tools, which are usually accessible via their own APIs, or via command line. The test generator exports the combinatorial model along with the generator options, into the the specific language of the external tool, or directly into the tool's APIs. Then, it waits for the generator to compute the test suite, and once ready, it passes it back to the REST service. If necessary, the generator also maps any parameter values back to the original CTWEDGE model format. Each external program needs its own translator.

The REST service accepts an HTTP GET or POST request with parameters as described in Tab. II. For sake of brevity, an example URL request to the web generator service is shown in Fig. 6.

So far, we interfaced CTWEDGE with ACTS [19] and CASA [15]. ACTS is accessed via it internal APIs whereas CASA is called via command line.

The resulting test suite is returned in CSV[6] format. The header line contains the parameter names, and each following line represents a single test, with parameter values. The web front-end allows to download CSV file to further local use, and shows the test suite in the browser by converting it into an HTML table. Conversion is straightforward and done client-side via Javascript.

[5]REST: Representational State Transfer
[6]CSV: Comma Separated Values


Not Secure | foselab.unibg.it/ctwedge/

CTWedge: Combinatorial Testing Web-based Editor and Generator

Load Example: Phone

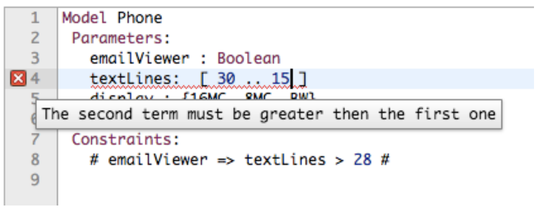
```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines:  [ 25 .. 30 ]
   display : {16MC, 8MC, BW}

 Constraints:
   # emailViewer => textLines > 28 #
```

Strength of Test Suite: 2

Generator: [✓ ACTS] CASA

Ignore Constraints

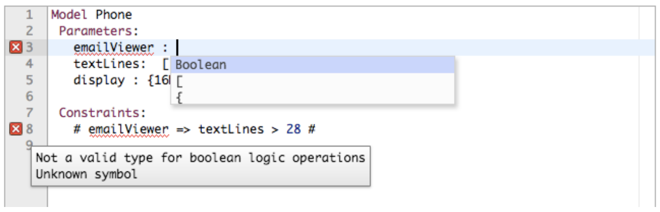
Fig. 4: CTWEDGE web editor



```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines:  [ 30 .. 15 ]
   display : {16MC, 8MC, BW}
 Constraints:
   # emailViewer => textLines > 28 #
```

The second term must be greater then the first one

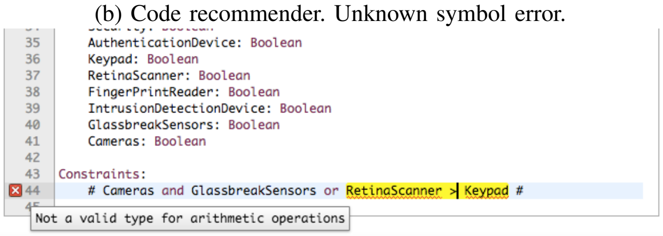
(a) Validation of Range parameter



```
Model Phone
 Parameters:
   emailViewer :
   textLines:  [
   display : {16
 Constraints:
   # emailViewer => textLines > 28 #
```

Not a valid type for boolean logic operations
Unknown symbol

(b) Code recommender. Unknown symbol error.



```
   AuthenticationDevice: Boolean
   Keypad: Boolean
   RetinaScanner: Boolean
   FingerPrintReader: Boolean
   IntrusionDetectionDevice: Boolean
   GlassbreakSensors: Boolean
   Cameras: Boolean

Constraints:
   # Cameras and GlassbreakSensors or RetinaScanner > Keypad #
```

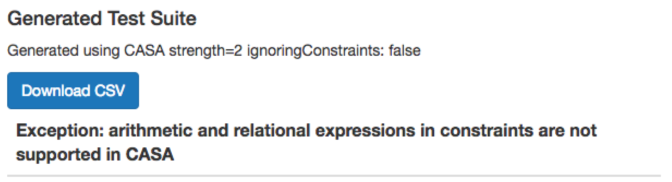
Not a valid type for arithmetic operations

(c) Validation of relational operations

Fig. 5: Examples of CTWEDGE validation errors

```
http://foselab.unibg.it/ctwedge.generator?
    ↪ model=Model%20Phone%20Parameter:%20...>
    ↪ 28%20#&strength=2&generator=acts&
    ↪ ignConstr=false
```

Fig. 6: Generator URL example



Generated Test Suite

Generated using CASA strength=2 ignoringConstraints: false

Exception: arithmetic and relational expressions in constraints are not supported in CASA

Fig. 7: Message of operations not supported in constraints for CASA generator tool

Arithmetic operations and relational operators ($>$, $<$, $\leq$, $\geq$) are not natively supported by CASA, and in presence of such constraints, an error is reported to the user, as in Fig. 7.

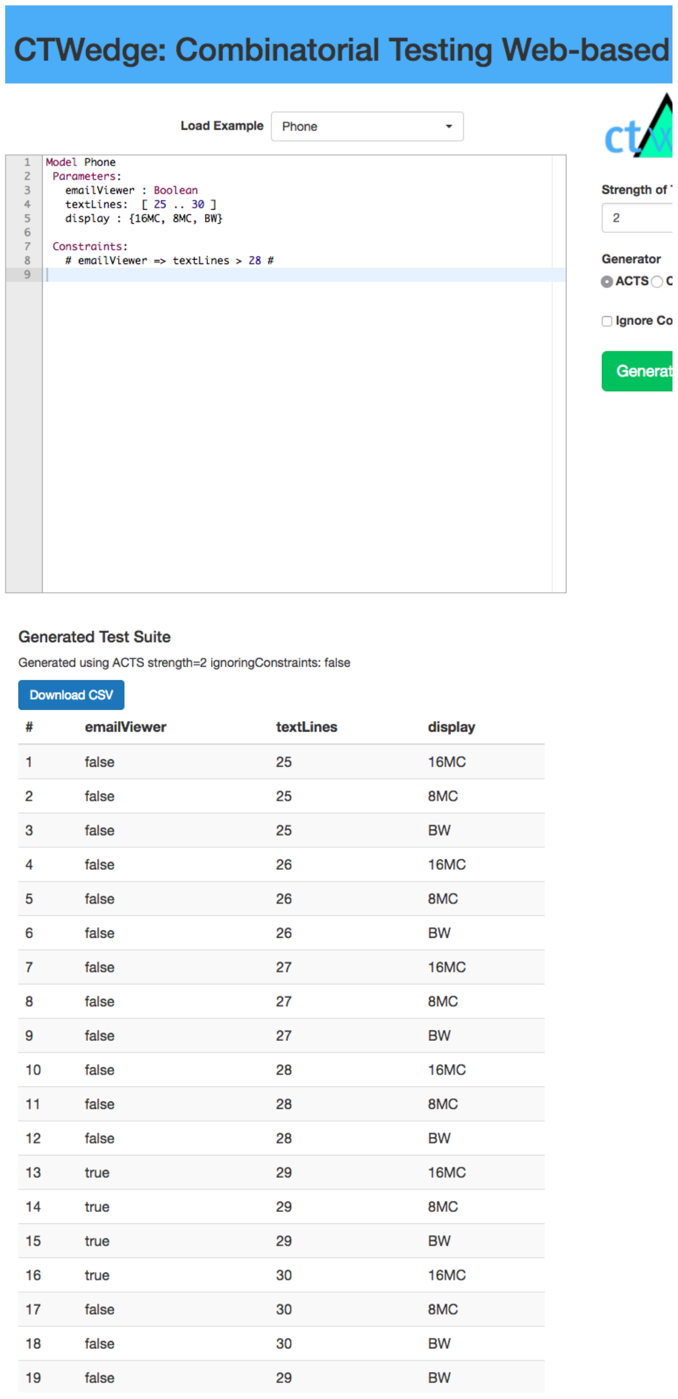
The HTML table showing the generated test cases is located below the editor on the same window. This location is less invasive than a brand new tab as it does not hide or replace the current combinatorial model in the editor. Despite it is not immediately visible to the user, who may think the output is hidden, we believe that it preserving the access to the current screen is the most important aspect to be preserved.

The generator is called by the editor by AJAX, with an asynchronous XmlHttpRequest.

A screenshot of how the generated test suite is presented to the user is shown in Fig. 8. It is shown as plain HTML table, with the possibility to download the data as CSV.

IV. Related Work

With the success of the SaaS pattern, web-based tools have rapidly gained popularity due to their portability and ease of use. There exist some web-based services also for combinatorial test case generation, each with its own peculiarities. SaaS tools, however, still represents a small number of all the available tools for test case generation: IDE plugins and desktop applications represent almost the totality of the



```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines:  [ 25 .. 30 ]
   display : {16MC, 8MC, BW}

 Constraints:
   # emailViewer => textLines > 28 #
```

Generated Test Suite

Generated using ACTS strength=2 ignoringConstraints: false

#	emailViewer	textLines	display
1	false	25	16MC
2	false	25	8MC
3	false	25	BW
4	false	26	16MC
5	false	26	8MC
6	false	26	BW
7	false	27	16MC
8	false	27	8MC
9	false	27	BW
10	false	28	16MC
11	false	28	8MC
12	false	28	BW
13	true	29	16MC
14	true	29	8MC
15	true	29	BW
16	true	30	16MC
17	false	30	8MC
18	false	30	BW
19	false	29	BW

Fig. 8: CTWEDGE visualization of the generated test suite

current tools. We looked for tools from the pairwise.org[7] and softwaretesters.net[8] tool catalogs, and from the web, to the best of our searching skills.

We compare the following five SaaS for CIT, listed in Tab. III:

[7]See http://www.pairwise.org/tools.asp

[8]See https://softwaretesters.net/zbxe/index.php?mid=downloadtool&category=4258006&sort_index=readed_count&order_type=desc

Tool	URL	Documentation
TestCover	https://testcover.com	https://testcover.com/sub/instructions.php (visible after registration)
CTWebClassic	http://alarcostest.esi.uclm.es/CombTestWeb/combinatorial.jsp	http://alarcostest.esi.uclm.es/CombTestWeb/stuff/usersManual.pdf
CTWebPlus	http://www.testcasegeneration.com or http://www.ctwebplus.com/	http://www.ctwebplus.com/stuff/userManual.pdf
HexaWise	https://hexawise.com/	https://hexawise.com/Hexawise_Introduction.pdf
PairWiser	https://inductive.no/pairwiser/	https://inductive.no/pairwiser/knowledge-base/

TABLE III: Tool resource links

- TestCover [7] is a commercial web-based combinatorial test case generator supported by Testcover.com, LLC, founded in 2003. The tool was also presented at IWCT 2016 [20].
- CTWeb Classic [5] is a free online tool for combinatorial testing and state machine test case generation, developed at University of Castilla-La Mancha (Spain).
- CTWeb Plus [6], an academic combinatorial test generation tool developed as improvement of CTWeb Classic. CTWeb Plus is now commercially supported.
- HexaWise [8], a commercial combinatorial test case editor and generator, launched in 2009 by Hexawise, Inc.
- PairWiser [9], a commercial web-based tool provided by Inductive AS. The online version was shut down January 15th, 2018. After that date, only the standalone application, for own-server installation, is available.

All tools are well documented, with examples and tutorials. Links of on-line editors and official documentation resources of these tools are shown in Tab. III

To compare the SaaS tools among them and w.r.t. CTWEDGE, we consider the following aspects, that we believe to be among the most relevant for a test engineer interested in using a web-based combinatorial test generation tool:

- **Language**. We look into the expressiveness of the accepted format for the combinatorial model in input. This evaluation includes:
  - Parameter definition: how the parameter types and values can be defined. For instance, a tool may support Boolean parameters or ranges of integers to express an enumerative made of all integers between two numbers.
  - Constraint format: if the constraints can be expressed as free combinations of logical and arithmetic operations among parameters, or have special formats, such as a set of forbidden tuples, a set of implications, or a set of if-then-else conditions.
  - Numeric Operations: if constant numbers and basic numeric operations (+, -, *, /) are allowed in the constraints and/or in the generated code of test cases.

	TestCover [7]	**CTWeb Classic** [5]	**CTWeb Plus** [6]	**HexaWise** [8]	**PairWiser** [9]	CTWEDGE
Language						
Parameter Definition	Enumerative	Enumerative	Enumerative	Enumerative, Ranges (via value expansion)	Enumerative	Boolean, Enumerative, Ranges
Constraints format	in DPB notation: via *blocks* (i.e., sets of allowed combinations)	as *if-then-else*	AND, OR, Else operators, not nested	invalid pairs (*if..then..*)	guided by select boxes with rich choice of operators	arbitrary formula
Numeric operators	✓(in PHP functions)	✗	✓	✓	✗	✓
State Machine support	✓	✓	✓	✗	✗	✗
Editing						
Web-based editor	text area	text fields and buttons + file upload	text fields, buttons, drawing area	text fields and buttons	text fields and buttons	text area
Model Import/Export	✓(Copy&Paste as text)	✓	✗	✗	✗	✓(Copy&Paste as text)
Helping facilities	✗	button-guided (no facilities to build input file to upload)	button-guided	button-guided	button-guided	content-assist, syntax highlight, in-line error reporting
Example Models	✓	✓	to be rebuilt from documentation file, not one-click loadable	✓	in the documentation	✓
Generation						
n-wise	pairwise	pairwise	pairwise	up to 6-way interaction + mixed strength	up to 3-way interaction + mixed strength	✓
Supported generators	All-pairs	AETG, PROW, All combinations, Each choice, Random, Bacteriologic	AETG, Pairwise, All-combinations, Each-choice, Comb, Random	not specified	not specified	ACTS, CASA
Export formats	HTML, WSDL interface	HTML, CSV	HTML	HTML, Excel, CSV, OPML	Excel, Jira issues	CSV, HTML
Generate test scripts	✓(for Selenium)	✓(custom)	✓(custom)	✗	✓(custom)	✗
Coverage visualization	✓	✗	✗	✓	✓	✗
Other information						
License	Commercial	Free	Commercial	Commercial	Commercial	Free
Registration	subscription required	optional	subscription required	subscription required	own-server installation	✗
Online storage	✓	✗	✓	✓	✓(on own server)	✗
Additional notes	Functions in PHP into constraints. Also accessible via WSDL interface.	Registration required for models with more than 5 parameters	Features a drawing area to represent states and transition of a state machine. The combinatorial model must be in the form of a state machine.	Has also a chart showing the interaction coverage after each test	Pairwiser online was shut down January 15, 2018. Available only for own-server installation. Allows to specify combinations to include in test suite.	–

TABLE IV: A comparison with other SaaS for CT

  - State Machine support: if the language supports an easy input of state machines, to generate combinatorial tests for their execution.
- **Editing**. We evaluate how simple is for the test-engineer to input the combinatorial model into the tool. This category includes the following aspects:
  - Web-based editor structure: how is the GUI of the web editor for writing the combinatorial model to be given in input to the tool; for example, if it is made by a single text area, or some buttons and text fields. Some tools use a single text area for the whole model, whereas some other tools use text inputs for individual parameters, reducing the need for parsing and text-highlighting.
  - Import and export models: how the models can

  be exported to the file system and imported. For example, a tool may allow importing a text file written with another editor.
  - Helping facilities: how the test engineer is guided in the input of the model in the web editor; for example with syntax highlighting, content assist, in-line error reporting, warning messages, or single text input fields to fill, and self-explanatory buttons to click.
  - Predefined example models: if there are examples of combinatorial models that can be easily loaded into the tool and executed to generate a test suite.
- **Generation**. We evaluate how the test suite generation is performed and how the output is presented to the user. This category includes the following aspects:
  - n-wise: which interaction strengths of the generated test suite are allowed
  - supported generators: which existing combinatorial test generation algorithms are supported
  - export formats: in which formats the output is made available to the test-engineer
  - Test-script generation: if there is a mechanism to allow the generated test vectors to be directly inserted into test cases written in custom code.
  - Coverage visualization: if there is indication (textual or with charts) of the coverage reached after the execution of each test in the generated test suite.
- **Other information**. We consider aspects about the accessibility of the tool, and related features. We look the following aspects
  - License: if commercial, free, or open source.
  - Registration: if it is mandatory, optional or not made available.
  - Online storage: if any data (input models, or output test suites) can be stored online.
  - Additional notes: any other additional information that we consider worth being noted.

Table IV compares the five tools and CTWEDGE according to all these aspects.

Concerning the web editor, while TestCover uses, as CTWEDGE, a single text area for combinatorial model input, all the others (CTWeb Classic, CTWeb Plus, HexaWise and PairWiser) feature a composer of combinatorial model guided by multiple selectors, text fields, and buttons. This approach of using buttons and text input fields, has the advantage of a quicker learning curve, and it does not need a language grammar, nor a parser, nor the helping facilities typical of text-based editors, such as auto-completion and syntax highlighting. However, it is not always the preferable way to input combinatorial models in the tool. In fact, the availability of a domain-specific language makes it possible to quickly and easily write, edit and copy-paste combinatorial input models, and export, translate or port them to other platforms and tools.

CTWeb Classic comes both with a guided editor and a form to upload a text file containing the input of the tool, written in a domain specific language. Regarding the textual way of proving input for test case generation, however, although CTWeb Classic and TestCover have a good documentation, they have no facilities to help the test engineer in writing models. CTWeb Classic does not have an online editor for its own language (as it comes with just a file upload button), and TestCover has a simple text area, lacking support for auto-completion, syntax-highlighting and all the features proper of an IDE.

All the tried tools offer support for test case generation with constraints, to be specified in their specific formats. TestCover even allows to specify custom functions - in PHP code - to express constraints [20].

TestCover and CTWeb (Classic and Plus) offer pairwise test case generation, that is very often the chosen interaction strength by test-engineers. For some applications, however, higher interaction strength is preferred. HexaWise supports up to 6-way interaction strength, while PairWiser up to 3-way. CTWEDGE is, instead, the only tool that does not pose limitation (in theory) on the interaction strength of the test suite. However, HexaWise and PairWiser come with the additional possibility to specify a mixed test suite strength, i.e., values of each single parameter may be covered with different strength.

Still none of the tools supports the Microsoft Language Server Protocol [17], a new common open protocol for language servers which provides programming language-specific features to source code editors or integrated development environments (IDEs). The main goal of the standard is to support programming in any given language independently of editors or IDEs. We plan to extend CTWEDGE in order to support LSP.

V. Future Work

There are several directions in which we plan to work.

a) Language extensions: Adding expressive power to the language for combinatorial models, in particular to express test seeds and goals, represents a direction for future work. Test seeds allow a tester to force the inclusion of certain test cases in the generated test suite [14]. Test seeds may be complete or partial. Test goals are extra-constraints: relations among parameters to be satisfied by at least one test in the generated test suites.

Some CIT approaches [18] introduce weights for parameter values. Weights reflect the importance of different values for a given parameter. The user can express further requirements over the solution involving weights. Even if the same constraints may be expressed in our language, it may become impractical. We plan to extend the CTWEDGE language in order to include user defined functions depending on parameter values. A possible function could be the weight of a parameter. Constraints and test goals could use such functions to express complex testing requirements.

b) Combinatorial model editor: To make the transition to CTWEDGE easier, a possible direction for future improvement is an importer that translates models written in other generator formats, into CTWEDGE language format.

Secondly, although CTWEDGE already follows the SaaS approach, there are still several features that could be added in order to offer new cloud-based services. The web site could offer a storage and persistence service, and the logged user could save his/her models on the CTWEDGE server and later recall the saved files and export/import them in other formats. The CTWEDGE could offer analysis services, like those presented in [21], able to find modeling faults. The user could use such techniques to check that the constraints are consistent, that there is no constraint implied by other constraints, and that the parameters and their values are really necessary. Also coverage measurement and analysis on the generated tests could be useful in order to check that they actually cover all the testing requirements.

Another future direction is the visualization of the individual t-tuples, as covered by each test in the test suite.

c) Test case generation: Another direction for future work regards test case generation. The server is configured to run CTWEDGE generator in a synchronous mode: the generator starts producing the test suite immediately, trying to serve all the requests. The server could have performance issues due to overloading in case for example there are many requests with large models. The web service could be improved by attaching a process scheduler and a load balancer. Test suite generation becomes therefore asynchronous also on the server, which queues the requests and makes the test suites available as soon as they are generated.

d) External generation tool support: An additional feature direction consists in the expansion of the support for test case generators, as PROW [22], PICT [23], HSST[9] [24], Medici [25], as well as an expansion on the customizations of each selected test generator tool, such as the selection of the test generation algorithm inside ACTS: IPOG [26], IPOG-D [26], IPOG-F [27], IPOG-F2 [27] and PaintBall [28]. The possibility to download the translated input file along with the executable command parameters for each of the generators allows further customization and therefore can be an interesting future extension of the tool.

e) Offline extensions: Combinatorial test case generation is used as a part of an automated process for application testing. Thus running the test generation tool off-line is needed, in certain scenarios, to ease interfacing with automated tools or with IDEs during development process. We therefore plan to release a version of the CTWEDGE editor as Eclipse plugin. Xtext, already generates an eclipse-based development environment providing editing experience known from modern IDEs, featuring a content assist, quick fixes, a project wizard, template proposal, outline view, hyperlinking, and syntax coloring.

VI. Conclusion

Generation of combinatorial test suites via web offers great advantages w.r.t. classical desktop applications. It is nowadays supported by a pool of tools, both open source and commercial. However, to the best of our knowledge, none of them has an integrated web editor support and a complete support of constraints. To work, they have their own language, with often just examples as unique description, and expect the user to write a file locally before uploading to the web-based tool. This process requires the test engineer to use another tool, may it be just a stand-alone text editor, with no auto-completion for that particular language, or a custom stand-alone editor with some grade of code recommendation.

To offer a complete SaaS environment for CIT, we have developed and deployed CTWEDGE. CTWEDGE was designed with three principles in mind: (1) installation-free and download-free, (2) ease of use, and (3) extensibility to support more generators. By using Xtext, we have defined a simple textual language which includes also the possibility to define complex constraints. Thanks to Xtext, a web editor can be easily deployed and it offers classical editing features like syntax highlighting and coloring, syntax validation, auto completion, and error messages. We have also developed a REST service that is able to generate CIT test suites exploiting third-party test generator programs. This test generators runs on the server and it can be called from the editor, thus providing a complete SaaS experience to the tester.

[9]HSST: Heuristic based on solution space tree

References

[1] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv*, vol. 43, no. 2, p. 11, 2011. [Online]. Available: http://doi.acm.org/10.1145/1883612.1883618

[2] J. Czerwonka, "Pairwise testing in real world," in *24th Pacific Northwest Software Quality Conference*, vol. 82, 2006.

[3] A. Gargantini and P. Vavassori, "Citlab: a laboratory for combinatorial interaction testing," in *Workshop on Combinatorial Testing (CT) In conjunction with International Conference on Software Testing (ICST 2012, April 17-21)*. Montreal, Canada: IEEE Computer Society, 2012, pp. 559–568.

[4] I. Menken, *SaaS - The Complete Cornerstone Guide to Software As a Service Best Practices Concepts, Terms, and Techniques for Successfully Planning, Implementing and Managing SaaS Solutions*. London, UK, UK: Emereo Pty Ltd, 2008.

[5] M. P. Usaola and B. P. Lamancha, "A framework and a web implementation for combinatorial testing," Informe técnico, University of Castilla-La Mancha, Tech. Rep., 2010. [Online]. Available: http://ctweb.abstracta.com.uy/stuff/wpCombinatorial.pdf

[6] [Online]. Available: http://www.ctwebplus.com/

[7] [Online]. Available: http://testcover.com/

[8] [Online]. Available: https://hexawise.com/

[9] [Online]. Available: https://inductive.no/pairwiser/

[10] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 129–139. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273482

[11] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, Jul 1997.

[12] C. Lott, A. Jain, and S. Dalal, "Modeling requirements for combinatorial software testing," in *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, ser. A-MOST '05. New York, NY, USA: ACM, 2005, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083281

[13] M. C. Golumbic and I. B.-A. Hartman, *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*. Springer Publishing Company, Incorporated, 2011.

[14] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960 – 970, 2006, advances in Model-based Testing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584906000401

[15] "CASA: Covering arrays by simulated annealing." [Online]. Available: http://cse.unl.edu/citportal/tools/casa/

[16] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 307–309. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869625

[17] [Online]. Available: https://github.com/Microsoft/language-server-protocol

[18] I. Segall, R. Tzoref-Brill, and E. Farchi, "Using binary decision diagrams for combinatorial test design," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*. ACM Press, 2011, p. 254. [Online]. Available: http://portal.acm.org/citation.cfm?doid=2001420.2001451

[19] "Advanced Combinatorial Testing System (ACTS)." [Online]. Available: http://csrc.nist.gov/groups/SNS/acts/

[20] G. B. Sherwood, "Embedded functions for constraints and variable strength in combinatorial testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on*. IEEE, 2016, pp. 65–74.

[21] P. Arcaini, A. Gargantini, and P. Vavassori, "Validation of models and tests for constrained combinatorial interaction testing," in *The 3rd International Workshop on Combinatorial Testing (IWCT 2014) In conjunction with IEEE International Conference on Software Testing (ICST 2014, March 31 - April 4)*, 2014, pp. 98–107.

[22] B. P. Lamancha, M. Polo, and M. Piattini, "PROW: A pairwise algorithm with constRaints, order and weight," *J. Syst. Softw.*, vol. 99, no. C, pp. 1–19, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2014.08.005

[23] [Online]. Available: https://osdn.net/projects/pictmaster/

[24] C. Nie, B. Xu, L. Shi, and Z. Wang, "A new heuristic for test suite generation for pair-wise testing," in *Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2006), San Francisco, CA, USA, July 5-7, 2006*, 2006, pp. 517–521.

[25] A. Gargantini and P. Vavassori, *Efficient Combinatorial Test Generation Based on Multivalued Decision Diagrams*. Cham: Springer International Publishing, 2014, pp. 220–235. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-13338-6_17

[26] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog: A general strategy for t-way software testing," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, March 2007, pp. 549–556.

[27] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, no. 5, p. 287, 2008.

[28] P. Ammann and J. Offutt, "Using formal methods to derive test frames in category-partition testing," in *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, Jun 1994, pp. 69–79.