Combinatorial Interaction Testing with CitLab

Andrea Calvagna
Dip. di Matematica e Informatica
University of Catania - Italy
Email: calvagna@cs.unict.it

Angelo Gargantini
Dip. di Ingegneria
University of Bergamo - Italy
Email: angelo.gargantini@unibg.it

Paolo Vavassori
Dip. di Ingegneria
University of Bergamo - Italy
Email: paolo.vavassori@unibg.it

Abstract—**In this paper the CitLab tool for Combinatorial Interaction Testing is presented. The tool allows importing/-exporting models of combinatorial problems from/to different application domains, by means of a common interchange syntax notation and a corresponding interoperable semantic meta-model. Moreover, the tool is a framework allowing embedding and transparent invocation of multiple, different implementations of combinatorial algorithms. CitLab has been designed tightly integrated with the Eclipse IDE framework, by means of its plug-in extension mechanism. It is intended to easy the spread of CIT testing both in industrial practice and in academic research, by allowing users and researchers to apply multiple test suite generation algorithms, each with its peculiarities, on the same problem models, and let them compare the results in order to select the one that best fits their needs, while alleviating from the pain of knowing all the different details and notations of the underlying CIT tools.**

Index Terms—**Combinatorial testing model, domain-specific language, Eclipse, Xtext.**

I. Introduction

Combinatorial interaction testing (CIT) has been an active area of research for many years. In a recent survey [12] Nie and Leung count more than 12 research groups that actively work on CIT area and many other groups and tools are missing in the count. In a previous survey, Grindal et al. [10] presented 16 different combination strategies, covering more than 40 papers. There are several web sites listing tools and approaches (like [13]), and publishing benchmarks and evaluations of tools and algorithms (like [7]). Being each of these tools the outcome of independent research and development processes, every one has its own user interface (some graphical, other textual), its own syntax, its own algorithms, and its own benchmarks (if any). Despite they all are designed to tackle the very same tasks, as a matter of fact,

there is not a common *abstract meta-model* to represent combinatorial problems with a precise semantics for parameters, values, their constraints, and related concepts, nor exists a common *modeling syntax* or model exchange format between tools.

The lack of a common syntax and semantic framework for CIT makes harder the research in this area w.r.t. the following issues:

- The *comparison* among tools and approaches, an activity very useful and used in research literature, is instead quite unreliable since every user must remodel in its own language and tool the case studies taken from another tool or from the literature, with possible errors and misunderstandings. If the researchers could exchange examples using such common syntax (like XML, or textual, or graphical), the comparison of techniques and approaches would be facilitated and more technically sound.
- While conceptually modeling an abstract CIT task, a research group may use a term with a meaning, while other groups use the same term with a slightly different meaning (for example *seed* or *partial test*).
- Limited assistance in writing the models: very often generation tools do not offer any editing capabilities (only a grammar and a parser) and are rarely integrated in any IDE for programming or design. Very often the formats accepted by algorithms and tools are quite hard to understand[1].
- All the CIT generation tools are strongly decoupled making difficult to switch the use from one tool to another and also difficult the reuse of information (e.g. custom settings) and data already inserted and available in one tool.

This situation is also an obstacle for practitioners from fully advantage from such many different CIT generation techniques available, other than slowing down the research in this area.

Sometimes, designers may prefer a tool or a technique because it provides an usable graphical or a web interface instead of searching for the best tool that suites their needs. For instance, one the most used tools ACTS [1], has a very nice graphical interface regardless the fact that the generation methods it supports (IPOG [11] and variants) may be not suitable for the design of particular combinatorial test suites since, for instance, its support for constraints is not as powerful as in others.

Similar difficulties rise for researchers willing to devise a new CIT technique and compare it with existing ones. In order to experiment a new test generation algorithm, a researcher should define a proper grammar and a parser, develop the libraries to manipulate the model data, and translate the benchmarks found in literature into the newly defined language. These activities can be error-prone and quite time consuming without adding any actual contribution to the real problem of generating “better” combinatorial tests.

[1]Consider for instance one of the best tools for Constrained CIT, CASA [8]. CASA accepts only constraints written as a conjunction of disjunctions over the symbols (CNF), in a quite difficult format to write for humans.

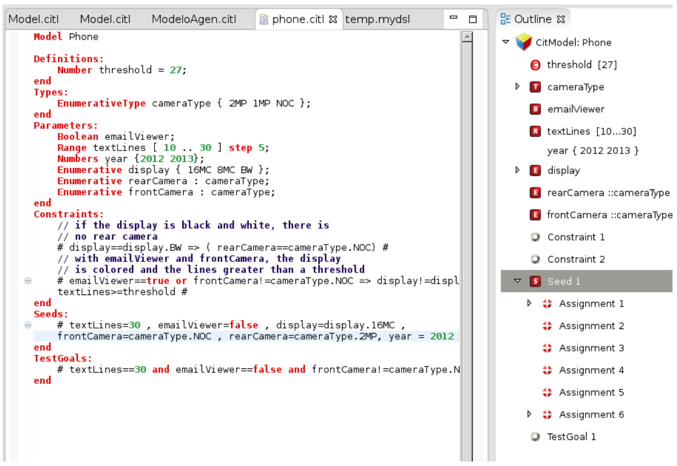
In this paper, we present CITLAB, a laboratory for combinatorial testing that tries to address all the aforementioned issues. CITLAB features:

- A rich abstract language with a precise *formal semantics* for specifying combinatorial problems.
- A concrete syntax with a well-defined *grammar* that allows practitioners to write models and researchers to share examples and benchmarks written in a "common" notation. Besides the concrete syntax given in XTEXT, CITLAB provides also an ANTLR grammar and an XMI interchange format for CIT models.
- A framework based on the Eclipse Modeling Framework (EMF) which provides tools and run-time support to (automatically) produce a set of Java classes for combinatorial models, along with a set of adapter classes and utility libraries that enable manipulating combinatorial problems in Java application using simple APIs. This allows developers to access combinatorial models inside their programs and tools.
- An editor integrated in the Eclipse IDE for editing combinatorial problems. The editor provides users with all the expected features in a modern programming environment like syntax highlighting, code completion, run-time error checking, quick fixes, and outline view.
- A simple EMF meta-model also for combinatorial test suites.
- A rich collection of Java utility classes and methods, specifically developed for combinatorial problems in CITLAB, which can be reused for manipulating combinatorial models and test suites. For instance, CITLAB provides utility methods for generating all the test requirements for a combinatorial coverage of strength t, a set of methods to check if a test suite satisfies all the requirements, and a set of methods for semantic validation of models and test suites.
- A framework for introducing new test generation algorithms which can be added to CITLAB as plugins. This allows researchers to develop new generation techniques and plug them in the framework without the burden of defining a grammar, a parser, an abstract syntax tree visitor, and so on.
- A framework for introducing code translators for importing and exporting models and tests to other notations based on Model to Text (M2T) or Model to Model (M2M) transformations. This could facilitate the use of CITLAB language as language for exchanging models and benchmarks.

II. USING XTEXT TO DEFINE CITLAB FRAMEWORK

The core of CITLAB is its language for defining combinatorial models. The development of a dedicated DSL (Domain Specific Language) and its corresponding editor, using the Xtext tool within the Eclipse framework, passed through the following five stages:

1) Grammar definition.
2) Configuration of the artifacts generator.



```
Model Phone

Definitions:
    Number threshold = 27;
end
Types:
    EnumerativeType cameraType { 2MP 1MP NOC };
end
Parameters:
    Boolean emailViewer;
    Range textLines [ 10 .. 30 ] step 5;
    Numbers year {2012 2013};
    Enumerative display { 16MC 8MC BW };
    Enumerative rearCamera : cameraType;
    Enumerative frontCamera : cameraType;
end
Constraints:
    // if the display is black and white, there is
    // no rear camera
    # display==display.BW => ( rearCamera==cameraType.NOC) #
    // with emailViewer and frontCamera, the display
    // is colored and the lines greater than a threshold
    # emailViewer==true or frontCamera!=cameraType.NOC => display!=displ
    textLines>=threshold #
end
Seeds:
    # textLines=30 , emailViewer=false , display=display.16MC ,
    frontCamera=cameraType.NOC , rearCamera=cameraType.2MP, year = 2012
end
TestGoals:
    # textLines==30 and emailViewer==false and frontCamera!=cameraType.N
end
```

Figure 1. A screenshot of the editor

3) Generation of the EMF/ecore metamodel for the language.
4) Generation of the DSL APIs and of the editor plugin.
5) Implementation of the scope and the validation rules.
6) Refinement of the text formatting and the content proposal provider.

At the end of this process, CITLAB has a grammar, a metamodel given in terms of EMF classes and relationships, a Java API useful to access programmatically combinatorial models and test suites, and a fully eclipse integrated editor.

The use of Xtext in the development process of CitL DSL eases its integration in the ui IDE context. User experience can benefit from the default functionality for code completion, syntax highlighting, syntactic validation, linking errors, the outline view and find references.

a) Syntax highlighting: it gives an immediate feedback about the use of keywords in different colors and fonts according to the category of terms. It is also useful to immediately detect errors to the user of the errors and typos.

b) Autocompletion: it helps the user to speedup the code editing by providing suggestions how to complete words and terms.

c) Outline view: it gives a graphical view of the structure of the combinatorial model.

d) Model Validation: it finds semantic errors at typing time.

e) Ide integration: CITLAB is integrated in eclipse since for the UI part (dialogs, ...) it uses the JFace/SWT eclipse API.

f) New project wizard: An inexperience user can start his/he modeling activity from a template of a simple combinatorial model.

III. USING CITLAB TO MODEL COMBINATORIAL PROBLEM

For details about the CITLAB language, see [9]. By means of a small example of a cell phone, we will explain in this section how to define a combinatorial model. A screenshot of the editor is given in Fig. 1. A CITLAB model consists in six parts: *Definitions*, *Types*, *Parameters*, *Constraints*, *Seeds*, and *TestGoals*.

In the *Definitions* section, the user can define numerical constants. For instance, the following statement introduces a constant with its value. Constants can be used in constraints, test goals, and seeds.

```
Number threshold = 27;
```

In the *Parameters* section the designer specifies the parameters (inputs) of the system. CITLAB language forces the designer to name parameters and to specify their types by listing all the values in their domain. Four kinds of parameter type are introduced:

- **Enumerative** for parameters that can take a value in a set of symbolic constants. Enumerative parameters are declared in the following way. For instance if the display of the cell phone can be colored (with 16 or 8 millions colors) or black and white, we introduce the following parameter.

  ```
  Enumerative display {16MC 8MC BW};
  ```

- **Boolean** for parameters that can be either true or false, which can be declared as follows. For instance if the phone can have an email viewer, the designer can introduce the following parameter.

  ```
  Boolean emailViewer;
  ```

- Numerical values in a **range** for parameters that take any value in an integer range. The user can also specify an integer step. For instance, if the phone has a number of lines between 10 and 30, but the designer want to test only this parameter every 5 values, he/she can write:

  ```
  Range textLines [ 10 .. 30 ] step 5;
  ```

  Note that the step can be omitted and in that case its value is 1, otherwise it must be a divisor of the difference between the two extreme values.
- A list of **Numbers** for parameters that take any value in a set of integers. It is like an enumerative, but mathematical comparisons and operations are allowed over these parameters.

  ```
  Numbers year {2012 2013};
  ```

Types can be implicitly introduced directly when declaring a parameter belonging to an *anonymous* type or they can be defined with their name in the Types section to be used in parameters declaration. For instance, a type can be defined as follows and this allows two parameters to share the same domain.

```
Types:
  EnumerativeType
        cameraType { 2MP 1MP NOC};
end
Parameters:
  Enumerative rearCamera : cameraType;
  Enumerative frontCamera : cameraType;
end
```

A. Constraints

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 [6]. Constraints were first described as being important to combinatorial testing in [5] and were introduced in the AETG system. 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 [6]). 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 CITLAB, we adopt the language of propositional logic 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.

In the CITLAB language all the constraints must be listed in a section called Constraints (and included between two # symbols). For instance, assume that the user wants to model the following two constraints

- if the display is black and white, then the phone cannot have any camera
- if the phone has a email viewer and front camera, the display is colored and the lines greater than the given threshold (defined as constant).

The following declarations can be added in the model in the Constraints section:

```
Constraints:

# display==display.BW => rearCamera==cameraType.NOC #

# emailViewer==true or frontCamera!=cameraType.NOC =>
   display!=display.BW and textLines>=threshold #
end
```

We assume that all the constraints must be satisfied by any test case, i.e., the constraints are conjoint with an implicit $\wedge$ operator. For a precise semantics of constraints and some formal definitions, see [9].

B. Seeds

The testers can also force the inclusion of their favorite test cases by specifying them as *seed* tests [2]. The seed tests must be included in the generated test set without modification. Since seeds represent tests the user has already executed or will execute in any case, the generation algorithm should take advantage of the seeds and avoid redundant coverage of interactions.

In CITLAB, seeds can be added in the Seeds section and can be expressed as a sequence of assignments as follows. For instance, the user wants to force the inclusion of the following combinatorial test, by writing in the model:

```
Seeds:
# emailViewer=false , display=display.16MC,
  frontCamera=cameraType.NOC, year = 2012,
  rearCamera=cameraType.2MP, textLines=30 #
end
```

C. Test Goals

CitLab allows the tester to introduce extra testing requirements by means of *test goals*. They must be considered in addition to the desired t-wise coverage. In fact, the user may be interested to test some particular critical situations or input combinations, for instance simple incomplete combinations, or more generic relations than simple combinations among parameters. For instance, if the user wants to be sure that the test suite contains at least a test in which at least one camera is missing and the display has at least threshold lines, he can write the following test goal:

```
# (rearCamera == cameraType.NOC or frontCamera ==
      cameraType.NOC) and textLines >= threshold #
```

Note that most tools do not support test goals and seeds, however we decided to include them in the language because one of the CitLab aims is to provide a standard common language capable to represent a rich variety of combinatorial testing concepts.

D. Model validation

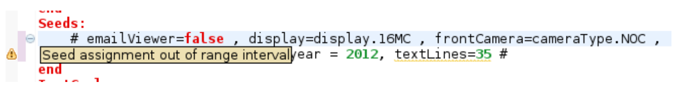
Besides editing capabilities (like syntax highlighting, autocompletion, and so on), Xtext provides several levels of validation for models of the defined language. The first level regards the syntactical validation done by the lexer and the parser, a cross link validation done by a linker and a concrete syntax validation done by the serializer that validates all constraints that are implied by a grammar. Besides these first three kinds of validation that are automatically introduced by Xtext, the user can specify additional constraints for the model by providing generator fragments. We have introduced the validation fragments for the following rules:

1) In each expression of kind $x = y$, where x is a parameter and y is a value, y must belong to the domain of x.
2) A seed must assign a (valid) value to each parameter.
3) No seed can violate any constraint.

The validation of the last requirement (3), requires the evaluation of constraints. This is performed by two classes, available in the APIs:

- Logic Evaluator: evaluates Boolean expression starting from the value of its operands.
- Arithmetic Evaluator: computes the integer value of a numeric expression.

The validation is performed run-time while the user types the model. If the validator finds an error in the model it generates an error message. The nature of the error is indicated in the error-log view of eclipse and the point in which the error occurs is marked in the editor. Fig. 2 shows how the editor checks the correctness of a seed.



```
Seeds:
    # emailViewer=false , display=display.16MC , frontCamera=cameraType.NOC ,
Seed assignment out of range intervalyear = 2012, textLines=35 #
end
```

Figure 2. Validation of seeds

IV. CitLab as extensible framework

Besides the definition of a language for combinatorial problems, together with its editor, meta-model, and Java API to manipulate combinatorial models, a further goal of CitLab is to introduce a framework for the definition and implementation of actual test generators and a set of exporters/importers to and from other languages to foster tools interoperability. In order to ease the development and deployment of such components that can extend its capabilities, CitLab relies on the extension techniques as defined by the Eclipse framework. In Eclipse, a framework or platform can accept new contributions as plugins by defining *extension points*. External contributors can add to the framework new plugins by implementing *extensions*. One can think of an extension point as a port – an entry point for other plugins to offer services. An extension is a plug that connects to the right port. An extension point defines a contract between the platform and the service provider introduced as plugin. The extension implementation is the actual service which will be added to the platform by using the plugin mechanism of Eclipse.

There are several benefits from this architecture. New plugins can be dynamically added and removed from the platform without recompiling them. Third-party tools can be easily added to the platform by registering them as extensions. A plugin includes some descriptive information and the platform extension point can decide how to use it. For instance, a plugin can declare to support a feature and the platform can decide if it is worth loading the extension or not. The development of a plugin is strongly decoupled with the development of the platform, making easy for third-parties to contribute to the framework.

An extension point can be used to introduce a Java interface which must be implemented by its extensions. The plugin that define the extensions must define a class implementing the required interface in order to extend the platform with new functionalities and register its extension into the platform. The platform will become aware of new functionalities and will be able to create and call instances of that class when needed. CitLab introduces four extension points listed in Listing 1.

These four points are mapped respectively to the following functions: the generation of a test suite with some embedded CIT algorithm; saving the computed test suite in some persistent (file) format, and importing/exporting a CIT model, with constraints, from/to some other notation.

CitLab introduces the four extension points together with the interfaces and the required methods listed in Tab. I.

A. CitLab *Architecture*

CitLab itself is a set of eclipse plugins. The overall architecture with the main components is shown in Fig. 3.

Listing 1. CitLab extension points

```
<extension-point id="importers"
      name="Importers"
      schema="schema/importers.exsd"/>
<extension-point id="exporters"
      name="Exporters"
      schema="schema/exporters.exsd"/>
<extension-point id="generators"
      name="Generators"
      schema="schema/generators.exsd"/>
<extension-point id="testsuiteexporters"
      name="TestSuiteExporters"
      schema="schema/testsuiteexporters.exsd"/>
```

Extension Point	Attribute and interfaces required
TestGenerator	`ICitLabTestGenerator` TestSuite generateTestsAndInfo(CitModel model, boolean ignoreConstraints, boolean ignoreSeeds, boolean ignoreTestGoals, int nWise, String generatorName); String Name String Algorithm boolean supportConstraints boolean supportSeeds boolean supportTestGoals

Exporter	`ICitLabExporter`
	void convertModel(CitModel citModel, Boolean constraintUse, int nWise);
	String Name
	String OtherToolLanguage

Importer	`ICitLabImporter` CitModel importModel (String path) throws NotImportableException; String Name String OtherToolLanguage String FileExtension

TestSuiteExporter	`ICitLabTestSuiteExporter`
	void generateOutput(TestSuite input, String FileName);
	String ExporterName
	String FileExtension

Table I
EXTENSION POINTS DEFINED BY CITLAB

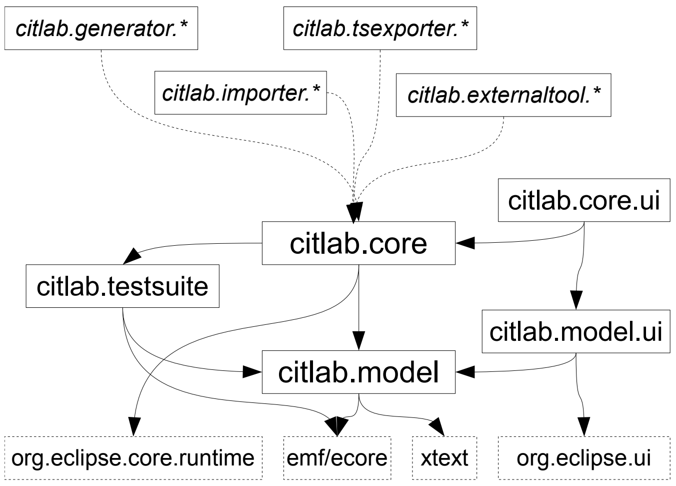
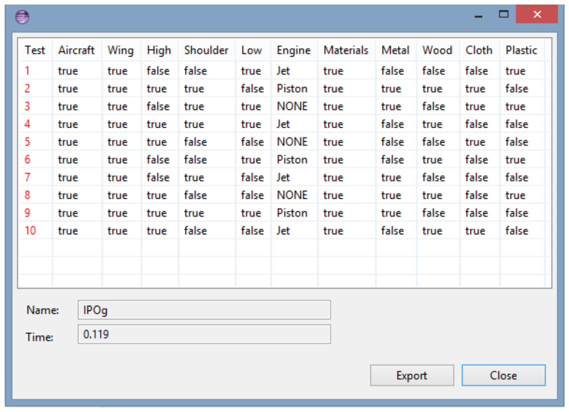


Figure 3. CITLAB Architecture



Test	Aircraft	Wing	High	Shoulder	Low	Engine	Materials	Metal	Wood	Cloth	Plastic
1	true	true	false	false	true	Jet	true	false	false	false	true
2	true	true	true	true	false	Piston	true	true	true	true	false
3	true	true	false	true	true	NONE	true	true	false	true	true
4	true	true	true	true	true	Jet	true	false	true	false	false
5	true	true	true	false	false	NONE	true	false	false	true	false
6	true	true	false	false	true	Piston	true	false	true	false	true
7	true	true	false	true	false	Jet	true	true	false	false	false
8	true	true	true	false	false	NONE	true	true	true	false	true
9	true	true	true	true	true	Piston	true	true	false	false	false
10	true	true	true	false	false	Jet	true	false	true	true	false

Name: IPOg
Time: 0.119

The project citlab.model contains the definition of the language and the Xtext utilities. The companion project citlab.model.ui contains the user interface for the language. The project citlab.testsuite introduces a simple meta-model, based on EMF, for tests and test suites. The project citlab.core introduces the extension points. Every plugin can extend the extension points defined in this component. The core plugin introduces also some APIs that can be used in order to implement specific generators. For instance, the following method:

```
Iterator<List<Pair<Parameter, String>>>
                         getTuples(CitModel m, int k)
```

can be used to list all the k-tuples for a model m, where each tuple is a `List` of pairs (`Parameter,String`). The utilities include methods for converting expression to CNF and checking if a test satisfies the constraints.

We use the following conventions for plugin names. We use citlab.importer.* projects to define importer components, which are able to translate combinatorial problems into the CITLAB language (for instance, we have defined an importer from feature models in [4]), while packages citlab.generator.* define algorithms and techniques for test generation (some are presented in the next section). The packages citlab.tsexporter.* define exporters for test suites (up to now CITLAB provides the tester with two pre-built exporters to Excel and to CSV format). We decided to define in citlab.externaltool.* generators which use external tools for test generation. They define an exporter from CITLAB combinatorial model to the notation of an external tool and then use that tool to perform the test generation.

All the code for CITLAB is available under the Eclipse Public License[2].

V. USING CITLAB FOR TEST GENERATION

CITLAB currently supports the following test generators, each defined as generator plugin.

- **AETG** is a plugin developed by us following the pseudo code for the greed algorithm of AETG published in [5].
- **IPO** is a plugin developed by us following the pseudo code for IPO published in [11]

[2]All the source code is available at http://code.google.com/a/eclipselabs.org/p/citlab/.

- **Random** is a simple random algorithm that adds new randomly built tests until all the n-wise combinations are covered. It avoids duplicates. It is useful as example for developing new plugins and for comparison with other new generation algorithms since it may represent the worst case in terms of test suite size in test generation.
- **ACTS** is ab external test generator tool developed by the NIST [1].
- **CASA** is an external tool for test generation based on simulated annealing [8].
- **ATGT_SMT** is an external tool combining heuristics and SMT solving [3].

	ACTS	**IPOG**	**ACTS**	**IPOF**	**Cover**	**CASA**
Model Name	*size*	*time*	*size*	*time*	*size*	*time*
GCC	25	0,56	24	0,44	22	3962
SPIN SIMULATOR	23	0,04	23	0,02	19	8,98
Dell Laptops 2009	36	3,60	38	3.92	51	70.9
Mobile Phone	12	0,01	10	0,01	9	0,76

Table II
GENERATOR COMPARISON (PAIRWISE COVERAGE TIME IN SECONDS)

CITLAB is able to query the eclipse installation and gather all the generator plugins available. To generate a test suite, the user simply selects the model and a test generator from those registered as plugin in CITLAB, and starts the job. When the test generator finishes, CITLAB shows the resulting test suite. Fig. IV-A shows a test suite. The user can export the test suite as excel or text file.

Note that test goals and seeds are not supported by all the generators. For this reason, before starting the test generation, CITLAB generation process ignores them if the chosen generator does not support. On the other hand, if the generator supports such construct, CITLAB permits the user to ask the test generation to consider them.

VI. USING CITLAB FOR RESEARCH PURPOSES

Using a common language and framework for test generators, allows researchers to have a set of benchmarks and to compare several tools in a fair and objective way. We have defined a rather big set of benchmarks taken from the literature and freely available now on CITLAB site. Researcher can download them and test inside CITLAB their algorithms. For instance, we have run some preliminary experiments using several examples and obtained the results shown in Table II[3].

REFERENCES

[1] Advanced Combinatorial Testing System (ACTS). http://csrc.nist.gov/groups/SNS/acts/.

[2] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.

[3] Andrea Calvagna and Angelo Gargantini. Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In Catherine Dubois, editor, *TAP*, volume 5668 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2009.

[4] Andrea Calvagna, Angelo Gargantini, and Paolo Vavassori. Combinatorial testing for feature models using citlab. In *International Workshop on Combinatorial Testing (IWCT) 2013*. IEEE, 2013.

[5] 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*, 23(7):437–444, 1997.

[6] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM Press.

[7] Charlie Colbourn. Covering array tables http://www.public.asu.edu/ ccolbou/src/tabby/catable.html.

[8] Covering Arrays by Simulated Annealing (CASA). http://cse.unl.edu/citportal/tools/casa/.

[9] Angelo Gargantini and Paolo 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)*, pages 559–568, Montreal, Canada, 2012. IEEE Computer Society.

[10] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

[11] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, September 2008.

[12] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv*, 43(2):11, 2011.

[13] Pairwise web site. http://www.pairwise.org/.

[3]Because all the models in the table present constraints, their testsuites have been generated using algorithms that support constraints.

APPENDIX

In this Appendix we describe the plan for the live demonstration.

A. *Install* CITLAB

We will show how to install CITLAB from the update site http://svn.codespot.com/a/eclipselabs.org/citlab/CitLabPlugins/.

B. *Using* CITLAB *to model combinatorial problems*

We will demonstrate how to model a combinatorial interaction problem, by creating a CITLAB model, using a case study as example (for instance a cell phone). We will show:

1) How to define parameters of different kinds (Boolean, enumerative, and integer range).
2) How to introduce types to reuse definitions of enumerative parameters.
3) What is the meaning and possible use of **constraints** and how to introduce them.
4) What is the meaning of **seeds** and how to introduce them.
5) What is the meaning of **test goals** and how to introduce them.

C. *Using* CITLAB *for tests generation*

We will show how to generate tests suites using several test generators (like ACTS, IPO, etc.). We will show how to export test suite to excel files.

D. *Extending* CITLAB

We will show how to download the CITLAB source (consisting of several Eclipse projects), how to organize the workspace, and to build the entire system. We will show how the designer can introduce a new test generation algorithm by extending CitLab with a small test generation technique (for instance a random generator).