

CITLAB: a Laboratory for Combinatorial Interaction Testing

Angelo Gargantini

Dip. di Ingegneria dell'Informazione e Metodi Matematici
University of Bergamo - Italy
Email: angelo.gargantini@unibg.it

Paolo Vavassori

Dip. di Ingegneria dell'Informazione e Metodi Matematici
University of Bergamo - Italy
Email: paolo.vavassori@unibg.it

Abstract—Although the research community around combinatorial interaction testing has been very active for several years, it has failed to find common solutions on some issues. First of all, there is not a common abstract nor concrete language to express combinatorial problems. Combinatorial testing generator tools are strongly decoupled making difficult their interoperability and the exchange of models and data. In this paper, we propose an abstract and concrete specific language for combinatorial problems. It features and formally defines the concepts of parameters and types, constraints, seeds, and test goals. The language is defined by means of XTEXT, a framework for the definition of domain-specific languages. XTEXT is used to derive a powerful editor integrated with eclipse and with all the expected features of a modern editor. Eclipse is also used to build an extensible framework in which test generators, importers, and exporters can be easily added as plugins.

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 [24] 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. [14] presented 16 different combination strategies, covering more than 40 papers. There are several web sites listing tools and approaches (like [26]), and publishing benchmarks and evaluations of tools and algorithms (like [11]). The most studied area in CIT is the test suite generation, where several research groups continuously challenge existing algorithms and tools in order to provide better approaches in terms of execution times, supported features, and minimality of the produced test suites. However, even considering only the test generation problem, the CIT community has failed so far in establishing solutions for the following issues:

- A *common abstract language* to model combinatorial problems with a precise semantics for parameters, values, their constraints, and related concepts.
- A common *concrete syntax* or exchange format for tools: All the CIT generation tools are strongly decoupled making difficult to switch the use from one tool to another and minimizing the reuse of information and data already inserted in one tool.
- The *comparison* among tools and approaches, an activity very useful and used in research literature, is therefore

quite unreliable since every user must redefine in its own language and tool the examples taken from another tool or from the literature, with possible errors and misunderstandings.

- 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 understand¹.

These issues may make difficult for practitioners the use of CIT generation techniques and may slow 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 [21] 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.

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 “standard” notation. Besides the concrete syntax given in XTEXT,

¹Consider for instance one of the best tools for Constrained CIT, CASA [12]. CASA accepts only constraints written as a conjunction of disjunctions over the symbols (CNF), in a quite difficult format to write for humans.

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.

Section II briefly introduces the CIT problem and the technologies used in our project. Section III introduces and defines the language, which includes parameters and types, constraints, seeds, and test goals. A powerful editor can be derived thanks to the XTEXT tool from the definition of the language, as explained in Sect. IV. Section V shows how the eclipse extension framework can be used to define an extensible platform in which test generators can be added as plugins. Also importers and exporters can be plugged in the platform in order to increase the inter-operability among tools. Section VI presents some plans for future work.

II. BACKGROUND

Combinatorial Interaction Testing (CIT), often called simply combinatorial testing or combinatorial testing design, aims at testing the software or the system with a selected combinations of input values or parameters. There exist several tools and techniques for CIT. Good surveys on the ongoing research in

CIT can be found in [24], [14], while an introduction on CIT and its efficacy in practice can be found in [20]. We assume in this paper that the reader is familiar with the CIT in general.

CIT can be considered a specific application domain for which developing a domain-specific language is worthwhile. Domain-specific languages (DSLs) are languages tailored to a particular problem domain [23] where they are very often used to model specific problems and solutions. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. DSL development is supported by several tools and technologies.

XTEXT [30] is a framework for development of programming languages and domain specific languages. The developer can describe his very own DSL using XTEXT simple EBNF grammar language and the generator will create a parser, an AST-meta model (implemented in EMF) as well as a full-featured eclipse text editor from that. The framework integrates with technology from eclipse modeling such as EMF, GMF, M2T and parts of EMFT. Development with XTEXT is optimized for short turn-arounds, so that adding new features to an existing DSL is a matter of minutes. Still sophisticated programming languages can be implemented. The APIs of the DSL and its specific editor are generated as eclipse plugins; this important feature permits the development of tools, related to the DSL, that are fully based on the eclipse-IDE integration. XTEXT generator uses the special modeling workflow engine (MWE2) to configure the generator.

XTEXT uses the lightweight dependency injection (DI) framework Google Guice to wire up the whole language as well as the IDE components created by its generator. Most parts of XTEXT are implemented as services. A service is an object which implements a certain interface and which is instantiated and provided by Guice. When Guice instantiates an object, it also supplies this instance with all its dependent services. This architecture makes XTEXT one of the most interesting framework for development of DSL.

The DSLs developed using XTEXT can be translated to other languages or notations. XTEXT 2.0 provides the possibility to integrate a Model to Text (M2T) code generator. The code generation is done with Xtend2 that substitutes the old template language XPand. Xtend2 is a statically-typed programming language which is tightly integrated with and runs on the Java Virtual Machine. It is the natural successor to Xtend which allows to have XPand template syntax as an expression, in fact it is more Java like than XPand, but also has some similarities. The introduction of Xtend2 makes the developer able to choose the model to model transformations approach in the code generation. It offers in fact a polymorphic dispatch, extension methods or the template syntax. Xtend files are directly compiled to Java source code, which means that the developer does not have to manage byte-code when he wants to debug a generation process.

III. DEFINITION OF A LANGUAGE FOR CIT PROBLEMS

The first goal of our project is the definition of a domain specific language for combinatorial problems. We identify four parts we want to model: the parameters and their types, the constraints among them, the seeds to be included in the final test suites, and further test goals defined by the user. Fig. 1 shows an example of combinatorial model, called Phone, a modified version of an example given in [8]. Constraints, seeds, and test goals are removed by the example for simplicity and they will be discussed later in the paper.

A. Parameters and their types

A model for a combinatorial problem consists in several parameters (at least 2) which can take values in their domain. To describe a combinatorial problem would be sufficient to specify the *number* of variables and their *cardinality*. Many papers, like [17], simply use the exponential symbolic notation x^y to model y parameters each of which can take x values. For heterogeneous alphabets, the notation is extended as $x_1^{y_1} x_2^{y_2} \dots x_n^{y_n}$ to model y_i parameters that take x_i values with $i = 1 \dots n$.

CITLAB language forces the designer to name parameters and to specify their types by listing all the values in their domain. Three kinds of parameter type are introduced:

- **Enumerative** for parameters that can take a value in a set of symbolic constants.
- **Boolean** for parameters that can be either true or false.
- **Numerical value** for parameters that take any value in an integer range.

Types can be defined with their name in the **Types** section to be used in parameters declaration or can be introduced directly when declaring a parameter belonging to an *anonymous* type. For instance, in Fig. 1 only one type is defined, namely **cameraType**. The type of **display** has no name and it is considered anonymous. Moreover, we allow the definition of parameters with a constant value. Note that all the types must have finite cardinality, and so only ranges are allowed for numerical parameters.

For instance, the hypothetical product line for smartphones of Fig. 1 supports three **display** options (16MC, 8MC, BW) and the presence of a text email viewer (parameter **emailViewer**). A phone may have two cameras, rear and front, which can be of two types (2MP or 1MP) or without a camera (NOC). The **textLines** denotes the number of lines in the emailViewer and it is an integer between 25 and 30. The parameter **threshold** is a constant with value 27, that may be used in other expressions (like constraints).

We prefer to require explicit parameter names to facilitate the modeling of real systems in which parameters have an actual meaning since they normally refer to actual system features. Furthermore, explicit names of parameters allow the specification of constraints and seeds as explained in the following. Note that the parameter names should not influence test generation algorithms.

Moreover, we decide to allow the use of parameter types with names to make more compact and more maintainable

Model Phone

Types:

EnumerativeType

cameraType { 2MP 1MP NOC};

end

Parameters:

Number threshold 27;

Boolean emailViewer;

Range textLines [25 .. 30] ;

Enumerative display {16MC 8MC BW};

Enumerative rearCamera **type** cameraType;

Enumerative frontCamera **type** cameraType;

end

Constraints: ... end

Seeds: ... end

TestGoals: ... end

Figure 1. A smartphone example

the models in case many parameters share the same domain. In the example in Fig. 1, two parameters **rearCamera** and **frontCamera** share the same type **cameraType**.

Definition of Parameters and Types. A CIT problem is modeled by a set P of n parameters p_i with $i = 1 \dots n$, where $n > 1$ and each parameter has *type* the domain D_i having m_i values, i.e., $D_i = \{d_1^i, \dots, d_{m_i}^i\}$. A parameter p_i with $m_i = 1$ is a *constant*.

Definition of Tests and Test Suite. A *test* or *test case* is a function that assigns to each parameter p_i a value in D_i . A *test suite* is a set of tests.

Sometimes tests are modeled as n -uple of values, assuming that the parameters are ordered. In this case the n -uple (v_1, v_2, \dots, v_n) fixes the values for n parameters p_1, p_2, \dots, p_n .

A test suite achieves the t -way interaction coverage (t is also called strength) if every combination of any t parameter values is tested at least once [24], [19].

Definition of t -way coverage. A test suite achieves the t -way *combination coverage* if for all the t -way combinations of the n parameters in P , every variable-values configuration is covered by at least a test.

Comparison with other approaches: It is apparent that the exponential notation $x_1^{y_1} x_2^{y_2} \dots x_n^{y_n}$ can be easily translated in the proposed notation without any loss of information. It suffices to introduce $\sum y_i$ parameters p_{ij} with range domain $[1 \dots x_i]$ with $i = 1 \dots n$ and $j = 1 \dots y_i$.

Note that sometimes parameters are referred as *factors* and their cardinality as *level* [3].

In AETG [22], parameters and their values are defined in the basic construct *relation* that is a table with columns for each input item, and rows for the values of each input item. *Input* items are called fields and represent the parameters. Each field can have a different number of values, each of which is

a symbolic string. The translation to the CITLAB language is straightforward. For instance, in AETG, the problem of Fig. 1 would be represented as the following table:

emailViewer	textLines	display	rearCamera	frontCamera
true	25	16MC	2MP	2MP
false	26	8MC	1MP	1MP
	27	BW	NOC	NOC
	28			
	29			
	30			

Sometimes (like in [21]) parameters are named P_i and their domains are number starting from 1 to the domain cardinality. Also in this case the translation is straightforward.

Further constructs for modeling parameters and their types we have found in literature are presented in Section VI.

B. 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 [10]. Constraints were first described as being important to combinatorial testing in [7] 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 [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 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.

Definition of Constraints. A constraint is a predicate P over the parameters, i.e., $P(p_1, \dots, p_n)$. We say that a test t satisfies a predicate P or it is a model of P and we write $t \models P$ iff P holds by substituting each parameter p_i in P with the value of p_i in t .

In the CITLAB language all the constraints must be listed in a section called **Constraints** (and included between two **#** symbols). For instance, the following constraint may be listed in the model of Fig. 1,

```
# emailViewer or frontCamera != NOC =>
    display != BW and textLines >= threshold #
```

It models the requirement that, if the phone has an email viewer or a front camera, then the display cannot be black and white and the lines of the email viewer must be greater than or equals to the threshold.

We assume that all the constraints must be satisfied by any test case, i.e., the constraints are conjoint with an implicit \wedge operator.

Definition of valid tests. Let C_j be the constraints of a combinatorial problem with $j = 1 \dots m$. A test t is *valid* if it is a model of the constraints, i.e., $\forall j \ t \models C_j$ or equivalently $t \models \bigwedge_j C_j$.

Due to the presence of constraints, not all the t -way combinations are coverable. We say that a combination is *not coverable* if there is no valid test that can cover it. A test suite still achieves the full coverage if it contains only valid tests and it covers all the coverable combinations.

Finding if there exists a model that satisfies the constraints, and building that model, is an NP-complete problem, since it can be translated to a SATisfiability problem. However, checking if a test is valid is linear with the dimension of the constraints and can be performed easily.

Note that the constraints may be *inconsistent*, i.e., it is impossible to find any test that satisfies them. Consistency checking of constraints is again a SATisfiability problem.

Definition of consistent constraints. Let C_j be the constraints of a combinatorial problem with $j = 1 \dots m$. The constraints are *consistent* if there exists at least a test that satisfies them, i.e., $\exists t \ t \models \bigwedge_j C_j$.

Comparison with other approaches: The CITLAB language is expressive enough to represent most constraints we found in examples and case studies presented in other papers. Note that, in order to deal with constraints, some methods require to remodel the original specification, while very few directly support constraints in CIT. Cohen et al. [10] found that just one tool, PICT [13], was able to handle *full* constraints specification, that is, without requiring remodeling of inputs or explicit expansion of each forbidden test case. Test generation in the presence of (general) constraints has become a very active research topic in the CIT arena [5], [8].

Most tools provide a limited support for constraints. For instance, AETG [7], [22] requires to separate the inputs in a way that they become unconstrained, and only simple constraints of type **if then else** (or **requires** in [10]) can be directly modeled in the specification. The translation of those templates into our logic is straightforward. For example the **requires** constraint is translated by an *implication*; the **not supported** to a *not*, and so on.

Many approaches [10], [15] allow constraints only in the form of forbidden configurations [16]. A forbidden combination would be translated in our model as a *not* statement. For instance, a forbidden pair $x = a, y = b$ would be represented by the following constraint:

```
# not (x = a and y = b) #
```

Requiring to explicitly list all the forbidden combinations can soon become impractical. As the number of input grows, the explicit list may explode and it may become practically unfeasible and error-prone to build it. For example, the model of mobile phones presented in Fig. 1 has the constraint that

“A front video camera requires also a rear camera”. This constraint would be translated into two forbidden tuples:

```
(frontCamera = 2MP, rearCamera = NOC);
(frontCamera = 1MP, rearCamera = NOC);
```

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

```
# frontCamera != NOC => rearCamera != NOC #
```

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

Other tools require the user to manipulate the definition of test parameters in such a way that unwanted combinations cannot possibly be chosen; either by splitting parameter definitions into disjoint subsets [28] or by creating hybrid parameters [29].

Moreover, we support constraints that not only relate two variable values (to exclude a pair), but that can contain generic bindings among variables. We believe that the translation from natural language to a fixed form of constraints (as in [9]) can be error prone, while allowing a more expressive constraint language reduces the likelihood of errors. Note that any constraint models an explicit binding, but their combination may give rise to complex implicit constraints [10]: implicit constraints do not need to be formalized in our language.

For example, a constraint may require $x \neq y$, another $y = z$. They model also the implicit constraint $x \neq z$ that in our approach does not need to be listed, since it is implied by the original two constraints.

Note that in our approach, the constraints must be satisfied by any test case we obtain from the specification, i.e., a test case is *valid* only if it does not contradict any constraint in the specification. Others [3] distinguish between forbidden combinations (*hard constraints*) and combinations to be avoided if possible (*soft constraints*). We consider only hard constraints, for now.

Our framework is generic with respect how the test generation deals with constraints. They could be considered only after the test suite has been generated, by deleting tests which violate the constraints and then generate additional test cases for the missing combinations. By this approach, any classical algorithm for CIT may be extended to support constraints. However, this is usable only if the number of missing combinations is small and all the constraints are explicitly listed.

C. Seeds

The testers can also force the inclusion of their favorite test cases by specifying them as *seed* tests [3]. The seed tests must be included in the generated test set without modification. CITLAB considers only *complete* seeds, i.e., seeds that assign a valid value to each parameter (except the constants, which can be left out the seed).

Definition of seed. Given a combinatorial problem with parameters p_i with $i = 1, \dots, n$, a *seed* assigns to each parameter p_i a value in D_i , except if D_i has cardinality 1.

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:

```
# emailViewer = true, textLines = 30, display = 16MC,
    rearCamera = 2MP, frontCamera = 1MP #
```

Note that some seeds may be invalid if they violate the constraints. Checking if a seed violates a constraint is easy and has linear complexity. In Section IV we will present how this check is performed in CITLAB.

Comparison with other approaches: Many approaches and tools support seeds, like AETG [3], [7], PICT[13], NSS [25], and t-tuples and IPOs [6]. Some approaches support also *partial* seeds, i.e., tests that have some parameters with unassigned values.

D. 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, i.e., *p*-wise assignments with $p < n$ and $p > t$ (being n the number of parameters). We assume that these requirements can be modeled as generic logical predicates, allowing the same syntax as for the constraints. Combinations that must be covered and that do not involve all the parameters are easily translated to test goals. For example, if the user wants to test the phone system of Fig. 1 with a specific combination of inputs, such that both *rearCamera* and *frontCamera* are NOC and *display* is 16MC, he is allowed to write the corresponding test goal as follows:

```
# rearCamera = NOC and frontCamera = NOC
    and display = 16MC #
```

However, test goals can express more generic relations than simple combinations among parameters the designer wants to guarantee to be covered. 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 = NOC or frontCamera = NOC)
    and textLines >= threshold #
```

Note that some test goals may never be covered, for they are either a contradiction or in contradiction with the constraints over the parameters.

Definition of test goals. A test goal is a predicate tg over the parameters p_i . A test goal tg is *consistent* iff there exists a test that can achieve it, i.e., formally $\exists t \models tg$. A test goal tg is *feasible* iff there exists a valid test that can achieve it, i.e., formally $\exists t \models \bigwedge_j C_j \wedge tg$.

Inconsistent test goals are never feasible. All the feasible test goals should be covered.

Definition of covered test goals. Given a set of feasible test goals TG , a test suite covers them if for each test goal $tg \in TG$ there exists at least test that makes tg true.

A test generation method may support or not extra test goals. Checking feasibility of test goals is an NP-complete problem, since it can be reduced to a SATisfiability problem (assuming the all the parameters have finite domain). In CITLAB we assume that a generator method, if it supports test goals, it will also be able to check if they are feasible. We may add in the future a feasibility checker of test goals based on the use of SAT or SMT solving as done in [4]. However, checking if a test suite covers a test goal is easy.

Comparison with other approaches: Test goals can be used to represent partial seeds, used for instance by AETG [7]. Partial seeds do not specify the value of each parameter, however they must be covered by the final test suite. Most tools, like AETG, complete the partial test cases by filling in random values for the missing fields and adding the completed seeds to the test suite.

IV. XTEXT-BASED EDITOR FOR CIT PROBLEMS

The development of a textual DSL and its corresponding editor, using XTEXT, passes through the following five stages from the definition of the DSL grammar to the creation of a fully eclipse integrated text editor:

- 1) Grammar definition.
- 2) Configuration of the artifacts generator.
- 3) Generation of the DSL APIs and of the editor plugin.
- 4) Implementation of the scope and the validation rules.
- 5) Refinement of the text formatting and the content proposal provider.

In order to develop a new DSL it is necessary to create a new XTEXT project where the practitioner describes his very own DSL using the XTEXT simple EBNF grammar language. The next operation consists of defining the language configurations of the code generator. The whole generator is composed by fragments (as listed in Fig. 2) for generating parsers, the serializer, the EMF code, the outline view, etc. One of the most important features that XTEXT offers is the translation of of a DSL grammar to an EMF meta-model. XTEXT generator uses a special DSL called MWE2 - the modeling workflow engine.

For the development of the CITLAB language was necessary to customize many generated artifacts:

JavaScopingFragment - Scoping: The auto generated scope-provider results fully operating for all the semantic and syntactical expression except for the assignments that present cross-reference. The CITLAB language ensures the correctness of its cross-reference domains using a custom AbstractDeclarativeScopeProvider, without this customization it would be possible to write inconsistent assignments.

Class	Generated Artifacts
EcoreGeneratorFragment	EMF code
XtextAntlrGeneratorFragment	ANTLR
GrammarAccessFragment	Grammar access
ResourceFactoryFragment	EMF resource
ParseTreeConstructorFragment	Serializer
JavaScopingFragment	Scoping
JavaValidatorFragment	Model validation
CheckFragment	Model validation
FormatterFragment	Code formatter
LabelProviderFragment	Label provider
OutlineNodeAdapterFactoryFragment	Outline node
TransformerFragment	Outline
JavaBasedContentAssistFragment	Content assist
XtextAntlrUiGeneratorFragment	Content Helper
SimpleProjectWizardFragment	Project wizard

Figure 2. XTEXT fragments and artifacts

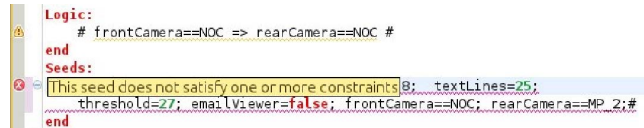


Figure 3. Validation of seeds

CheckFragment - Model validation: XTEXT provides several levels of validation for 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 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:

- 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. 3 shows how the editor checks the presence of inconsistencies between seeds and constraints while the user writes them.

Editor-Contributes: Once the grammar, the meta-model of the language, and the validation rules are defined, XTEXT

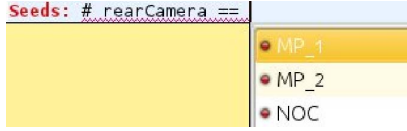


Figure 4. Content assist

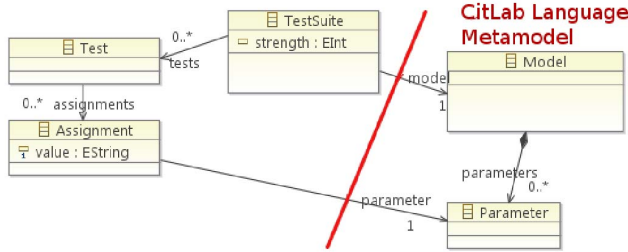


Figure 5. A meta model for Tests Suites

generates an eclipse-based development environment providing editing experience known from modern IDEs. The editor provides the developer with an IDE integrated in eclipse for writing combinatorial models in the CITLAB language. The editor features a content assist, quick fixes, a project wizard, template proposal, outline view, hyperlinking, and syntax coloring. For instance, the content assist shown in Fig. 4 helps the user to write the seeds and constraints while he types.

Test Suite meta-model and Java utils: CITLAB introduces also a simple meta-model for tests and test suites. The meta-model is shown in Fig. 5. which reports also a fragment of the language meta-model. A *TestSuite*, which refers to a *Model*, contains several *Tests* which can contain several *Assignments*. Each assignment gives a String value to a *Parameter*. We have added also the code to check the validity of the assignments and of the tests.

We have developed several classes and methods capable to perform routine tasks like generating all the test requirements for a given strength, checking if a test or a seed violates some constraints, and checking if a test goal is covered by a given test suite.

V. AN EXTENSIBLE FRAMEWORK FOR COMBINATORIAL TEST GENERATION

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

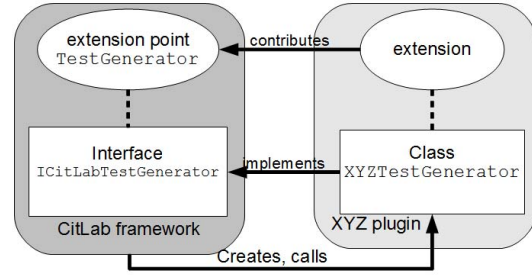


Figure 6. TestGenerator extension point and corresponding extension

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.

Very often an extension point introduces a Java interface which must be implemented by the extensions. As depicted in Fig. 6, CITLAB introduces an extension point called *TestGenerator* which defines the interface *ICitlabTestGenerator* declaring the methods necessary for any test generator. A plugin (XYZ in Fig. 6) must define a class (*XYZTestGenerator*) implementing the required interface in order to extend the platform with new test generators and register this extension into the platform. The platform will become aware of new generator and will be able to create and call instances of that class when needed.

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

All the code for CITLAB is available under the Eclipse Public License².

A. Test generation plugins

Any test generator plugin must introduce a class that implements the interface *ICitLabTestGenerator*. A test generator must declare if it supports the constraints, the seeds and the test goals (*accept* methods). Moreover, it must define a method that actually computes the test suite for the n-wise coverage of a given model (*generateTests* method). The method

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

Extension Point	Interface Methods
TestGenerator	ICitLabTestGenerator TestSuite generateTests(Model m, int nWise) boolean acceptConstraints(List<Constraint> c) boolean acceptSeeds(List<Seed> s) boolean acceptTestGoals(List<TestGoal> tg)
Exporter	ICitLabExporter void export(Model m) boolean acceptConstraints(List<Constraint> c) boolean acceptSeeds(List<Seed> s) boolean acceptTestGoals(List<TestGoal> tg)
Importer	ICitLabImporter Model import(Reader r)
TestSuiteExporter	ICitLabTestSuiteExporter void save(TestSuite ts, String file)

Table I
EXTENSION POINTS DEFINED BY CITLAB

will be invoked by the CITLAB user interface upon an user request. If an extension does not support some model features, like constraints for instance, the CITLAB user interface will prompt the user and will suggest to strip the model from that feature in order to continue with the test generation. Any test generator can decide to partially support a feature. For instance, a test generator may support only constraints having a particular pattern (like forbidden tuples or CNF). In this case, its method *acceptConstraints* will examine the constraints and it will decide if they conform to the requested pattern.

As proof of concept, we have implemented a small random test generator. It randomly generates tests cases until all the required tuples are covered. It does not support constraints neither test goals. However, it supports seeds. The pseudo code is reported in Fig. 7. Any realistic test generator should outperform this random generator.

B. Translation to and from other notations and tools

CITLAB introduces also two extension points for importing and exporting models from and to other notations and tools (as reported in Tab. I) By defining exporters and importers, the researchers could use the CITLAB language as a sort of *pivot* language. In DSLs, a pivot language can be used for exchanging models in several notations employed by different tools.

A possible way to define importers and exporters is to use Model to Text (M2T) or Model to Model (M2M) transformations. To this purpose, Xtend 2 can be used. For instance, we have defined an exporter to the CASA [12] language by defining a model to text transformer in Xtend. The transformer is translated to a Java class which implements the interface required by the exporter extension point. Inside the transformer, the designer can use polymorphic dispatching of template definitions and easily navigate through models. For instance, a fragment of the transformer for CASA is reported in Fig. 8 in which the size of a domain is converted to a string.

Transformations could also be defined at higher level, between meta-models (if provided by the target notations). In this

Require: *model* with *seeds* loaded

Require: *n* strength

```

// compute requirements using library methods
requirements ← allRequiredTuples(model, n)
// load seeds (if any) and discard covered requirements
for all s ∈ seeds do
  for all r ∈ requirements do
    if s covers r then
      // remove r from the requirements
      requirements ← requirements \ {r}
    end if
  end for
end for
testsuite ← seeds
// generate tests for uncovered requirements
while requirements ≠ ∅ do
  randomly generate a testcase
  toAdd ← false
  for all r ∈ requirements do
    if testcase covers r then
      requirements ← requirements \ {r}
      toAdd ← true
    end if
  end for
  if toAdd then
    testsuite ← testsuite ∪ {testcase}
  end if
end while
return testsuite

```

Figure 7. Random generation of a test suite

```

...
def getSize(Parameter param){
  switch (param) {
    Enumerative:
      "<((param as Enumerative).type.elements.size>""
    Boolean :""2""
    Number :""1""
    Range:
      "<(((param as Range).end-
        (param as Range).begin) as Integer).toString>""
  }
}

```

Figure 8. A fragment of the XTEXT exporter to CASA

case, the meta-model provided by CITLAB for combinatorial problems, could be used as *pivot meta-model* by providing suitable model to model transformations in order to allow tools interoperability. According to the view presented in [2], a pivot meta-model of a given formalism or language L is intended as a platform independent modeling language which abstracts a certain number of general concepts about L . The integration among tools supporting L can be achieved by providing, for the notation L' (a dialect of L) of each tool $T_{L'}$, a meta-model - seen as a platform specific modeling language - and model transformations to the pivot and from the pivot to the L' -meta-model. Hence, the meta-model of the notation L^i of a tool T_{L^i} can be linked to the meta-model of the notation L^j of another tool T_{L^j} by the composition of the two transformations from L^i -meta-model to the pivot and from the pivot to the L^j -meta-model. In this way, the interoperability between tools T_{L^i} and T_{L^j} is achieved by translating PSM (Platform-specific Model) models written in L^i to L^j and vice versa.

C. Test suite exporters

The third extension point is *TestSuiteExporter* that allows designers to add new plugins for exporting the test suite generated into files in specific formats. The plugin must introduce a class that implements the interface `ICitLabTestSuiteExporter`. In CITLAB we have already implemented an exported towards the Microsoft Excel format.

VI. FUTURE WORK

a) Language extensions: There exist approaches that introduce special constructs, not supported by the current version of the CITLAB language, for modeling complex situations in combinatorial models, beyond the basic parameters, values and constraints.

For instance, [22] describes the notions of compound values, auxiliary aggregates, and field groups. A compound is a set of values for fields and compound values reflect closely linked fields in a model. A compound field can be used to gather any number of linked values together into a set (or tuple). Compounds are useful when the interaction coverage is not required within the values in the compound, but it is needed with other parameters.

Auxiliary aggregates are used to link several combinatorial models since they capture parameters and values that are common between different test spaces. Field groups support optional groups of values. This means that, if any field from the group is present, then all fields from the group must be present; however, the entire group is optional.

PICT [13] introduces hierarchies of parameters. This scheme allows for certain parameters (at the bottom of the hierarchy) to be combined with a given strength first and that product is then used for creating combinations with parameters on upper levels of the hierarchy. This is a useful technique which can be used (1) to model test domains with a clear hierarchy of test parameters i.e., API functions taking structures as arguments and UI windows with additional dialogs or

(2) to limit the combinatorial explosion of certain parameter interactions.

A similar notion of sub-attributes is presented in [18]. It allows the user to express compound parameters, that are parameters associated with sub-parameters. For instance, consider a parameter A that has sub-attributes X , Y , and Z . The tester may decide to treat sub-parameters in different ways. For instance, sub-parameters may be considered as real parameters. In the example, the three parameters $A.X$, $A.Y$, and $A.Z$ would be added to the problem instead of the single parameter A .

Most of these constructs could be translated in suitable constraints also in the CITLAB language, but it could be useful to introduce some primitive constructs in the language in order to directly deal with these other concepts. This would make more compact the notation and test generator tools could exploit the new constructs in order to generate efficiently tests covering them. However, the semantics of these constructs should be well-defined in order to avoid misinterpretations. Sometimes (as in [18] for sub-attributes), it is left to user to choose the desired meaning of special notations.

b) Other types of constraints: In our approach, the constraints must be satisfied by any test case we obtain from the specification, i.e., a test case is *valid* only if it does not contradict any axiom in the specification. In [3] the authors introduce the concept of (explicit) *soft constraints*: they use a method to avoid tuples if possible. In this paper we consider only *hard constraints*: a test is valid only if it satisfies all the constraints (explicit and implicit as well). It could be interesting to introduce soft constraints also in the CITLAB language, although giving a formal semantics of such constructs may be rather complex.

Some CIT approaches [27] 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 CITLAB language in order to include user defined functions that are defined over the parameters and have value 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.

c) Seeds and test goals: Some tools introduce the concept of *partial seeds* which can be easily translated into our test goals as already explained. Other approaches classify partial seeds between valid, partially valid, and invalid [27]. Valid partial seeds are seeds that can be extended only into valid test cases. Invalid partial seeds cannot be extended to valid tests. Partially valid partial seeds are neither valid nor invalid, i.e., there is a valid extension but there is an extension which is invalid. The same concept can be formalized if the partial seeds are represented by test goals. A test goal is *invalid* if it contradicts the constraints. Formally if C_i are the constraints, and tg is a test goal, tg is invalid iff $\bigwedge C_i \rightarrow \neg tg$ or, equivalently $tg \rightarrow \neg \bigwedge C_i$. A test goal is *valid* iff

$tg \rightarrow \bigwedge C_i$, i.e., any model of tg satisfies also the constraints. A test goal is *partially valid* iff $\exists m(m \models \bigwedge C_i \wedge tg)$ and $\exists m(m \models \neg \bigwedge C_i \wedge tg)$. Checking the validity of a test goal is equivalent to the SATisfiability problem.

d) *Validation of constraints and test goals by SMT solvers*: As already noted, test goals may introduce further requirements that can be never satisfied given the types of the parameters and the constraints among them. On the other hand, also constraints may be inconsistent and never admit any solution. Finding tests covering test goals and constraints is equivalent to finding solutions to propositional logic formulas, which is an NP-complete problem. However, practical methods and tools (SAT solvers) exist that are very fast in finding models for most common proposition logic formulas. Recent work has extended the SAT solver algorithms to work with propositions containing arithmetic expressions; these are the SMT solvers. We plan to add a validator of constraints and test goals based on the use of SMT solvers in a similar way as done in [4] for test generation.

VII. CONCLUSIONS

We have presented a laboratory called CITLAB for combinatorial interaction testing. It introduces a language which is based on several formally defined concepts and it is defined as DSL by using the XTEXT methodology. The concepts are: parameters and types, constraints, seeds, and test goals. This language becomes endowed with a powerful editor and a set of Java APIs that allow the manipulation of combinatorial models and tests. CITLAB provides also an extensible framework based on the extension technique of eclipse where new test generators, exporters, and importers from other tools can be added as plugins. As a proof of concept we have already implemented a simple random test generator and an exporter to another tool using a model to text transformation.

Acknowledgments

We would like to thank Paolo Arcaini for the valuable comments on the paper.

REFERENCES

- [1] Advanced Combinatorial Testing System (ACTS). <http://csrc.nist.gov/groups/SNS/acts/>.
- [2] J. Bézivin, H. Brunelière, F. Jouault, and I. Kurtev. Model engineering support for tool interoperability. In *The 4th Workshop in Software Model Engineering (WiSME'05)*, Montego Bay, Jamaica, 2005.
- [3] 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.
- [4] 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.
- [5] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010. springer.
- [6] Andrea Calvagna and Angelo Gargantini. T-wise combinatorial interaction test suites construction based on coverage inheritance. *Software Testing, Verification and Reliability*, Special issue on Model Based Testing, 2011. JohnWiley&Sons.
- [7] 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.
- [8] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. *IEEE Transactions on Software Engineering*, to appear, 2008.
- [9] M.B. Cohen, M.B. Dwyer, and J. Shi. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference-Practice and Research Techniques (TAIC PART)*, London, September 2007, pages 121–130, 2007.
- [10] 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.
- [11] Charlie Colbourn. Covering array tables <http://www.public.asu.edu/~ccolbou/src/tabby/catable.html>.
- [12] Covering Arrays by Simulated Annealing (CASA). <http://cse.unl.edu/citportal/tools/casa/>.
- [13] J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, 2006.
- [14] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab.*, 15(3):167–199, 2005.
- [15] Alan Hartman. Ibm intelligent test case handler: Whitch, <http://www.alphaworks.ibm.com/tech/whitch>.
- [16] Alan Hartman. *Graph Theory, Combinatorics and Algorithms Interdisciplinary Applications*, chapter Software and Hardware Testing Using Combinatorial Covering Suites, pages 237–266. Springer, 2005.
- [17] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.
- [18] R. Krishnan, S. Murali Krishna, and P. Siva Nandhan. Combinatorial testing: learnings from our experience. *ACM SIGSOFT Software Engineering Notes*, 32(3):1–8, 2007.
- [19] D.R. Kuhn, R.N. Kacker, and Y. Lei. Practical combinatorial testing. Special publication, NIST, 2010.
- [20] R. Kuhn, R. Kacker, Yu Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94–96, aug. 2009.
- [21] 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.
- [22] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.
- [23] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, December 2005.
- [24] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11, 2011.
- [25] Changhai Nie, Baowen Xu, Liang Shi, and Ziyuan Wang. A new heuristic for test suite generation for pair-wise testing. In Kang Zhang, George Spanoudakis, and Giuseppe Visaggio, editors, *SEKE*, pages 517–521, 2006.
- [26] Pairwise web site. <http://www.pairwise.org/>.
- [27] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. page 254. ACM Press, 2011.
- [28] G. Sherwood. Effective testing of factor combinations. In *Proceedings of the Third International Conference on Software Testing, Analysis and Review*, pages 133–166, Washington DC, 1994.
- [29] A. W. Williams and R. L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Seventh International Symposium on Software Reliability Engineering (ISSRE iE'96)*, 1996.
- [30] Xtext. <http://www.eclipse.org/xtext/>.