Constraint-Based Automated Generation of Test Data

Hans-Martin Adorf and Martin Varendorff

Mgm Technology Partners, Frankfurter Ring 105a,
80807 München, Germany
{Hans-Martin.Adorf,Martin.Varendorff}@mgm-tp.com

Abstract. We present a novel method for automatically generating artificial test data that are particularly suited for testing form-centric software applications with several thousand input fields. The complex validation rules for user input are translated to a constraint satisfaction problem (CSP), which is solved using an off-the-shelf SMT-solver. In order to exert pressure onto the software under test, the generated test data have to incorporate extreme and special values (ESVs) for each field. The SMT-solver is aided by a sophisticated graph-based cluster algorithm and by other heuristic methods in order to reduce the complexity of the CSPs. With further optimizations, the test data generator now routinely generates a complete set of test data records for large form-centric applications within less than two hours. The test data generator described here is operationally being used for automated tests of form-centric Web-applications, within an iterative development process emphasizing very early testing of software applications.

Keywords: automated test data generation, constraint satisfaction problems, form-centric software applications, functional testing, satisfiability modulo theories, software quality assurance.

1 Introduction

The construction of comprehensive test data sets for large software applications is a complex, time consuming, and error-prone process. The sets have to include valid data records in order to support positive functional tests, as well as invalid data for negative tests.

In order to exert pressure onto the application one needs appropriate test coverage of the space of possible input data. Appropriate coverage requires seeking challenging input values for input fields, such as extreme or otherwise special values (ESVs), while simultaneously fulfilling all required validation rules for the input data, or, conversely, explicitly violating some of them. In order to limit the test execution time, the size of the test data set (i.e. the total number of test data records) should be small, which forces one to incorporate as many ESVs as possible into each test data record.

Large applications may require thousands of input values, which are subject to a similar number of validation rules. Test data sets have to be generated frequently. Therefore the automation of the entire data generation process is not only highly desirable, but a necessity for business-critical applications. This paper presents how we solved the problem of generating high-quality artificial test data sets that are mainly used within an efficient automated quality assurance process.

2 Testing Form-Centric Software Applications

Below we will concentrate on the quality assurance of "form-centric" software applications. The main purpose of such an application consists in providing users with "free-text" fields on forms for data entry. The input to a form-centric application ought to be validated before it is transferred to a processing system. This validation assures that each field entry is syntactically correct (single-field validation), and that the combination of entries into different fields is consistent (cross-field validation). The combination of definitions of fields and of validation rules is called a "validation rule-base".

A prime example of a large form-centric application is software supporting a tax declaration, where the taxpayer must enter names, addresses, earnings, calendar dates, etc. into free-text fields. Tax-related applications are particularly demanding, since they may contain up to two dozens of forms, some of which may occur in several instances (e.g. one form per child). Each form contains many fields, which in addition can be repeated (e.g. a list of deductibles). As stated above, the number of accompanying validation rules usually has the same order of magnitude as the number of fields, which can be in the range of several thousands.

Another example of a form-centric application is software supporting an applicant for an insurance policy, or an agent acting as an intermediary between the applicant and an insurance company.

2.1 The Test Process

Form-centric applications, as any other software application, must be tested before being deployed in the field. In our case, the high quality demands of our customers have so far been met by executing intensive manual tests, which, due to the large number of fields, are very tiring and costly. Over time, in order to save labor and to reduce costs, manual tests are replaced, as far as possible, by automated tests. For the automation of functional tests mgm has developed a test framework called jFunk [1].

At mgm we follow a software development life cycle emphasizing early and frequent testing of the applications under development (figure 1). A development cycle typically spans across several months. Within such a cycle, stable versions of the software are regularly produced in iterations, and they are tested mainly using automated functional tests. Test results are fed back into the development of the next iteration in order to prevent a build-up of defects within a cycle. The duration of the iterations decreases towards the end of a cycle. Accordingly, the frequency of executing automated tests increases, which requires a timely creation of test data sets.

2.2 Requirements for the Test Data Generation

In this section we will take a closer look at the requirements for, and the complexity of, the task of creating test data for a large form-centric application.

The validation rules in a rule base vary in complexity. Simple ones only check the presence of values in one or more fields. Others consist of equalities, inequalities, or disequalities between the values in two or more fields. Even more complex rules use functions of field values within numerical predicates. The most complex rules combine all types of conditions within sizable Boolean expressions.

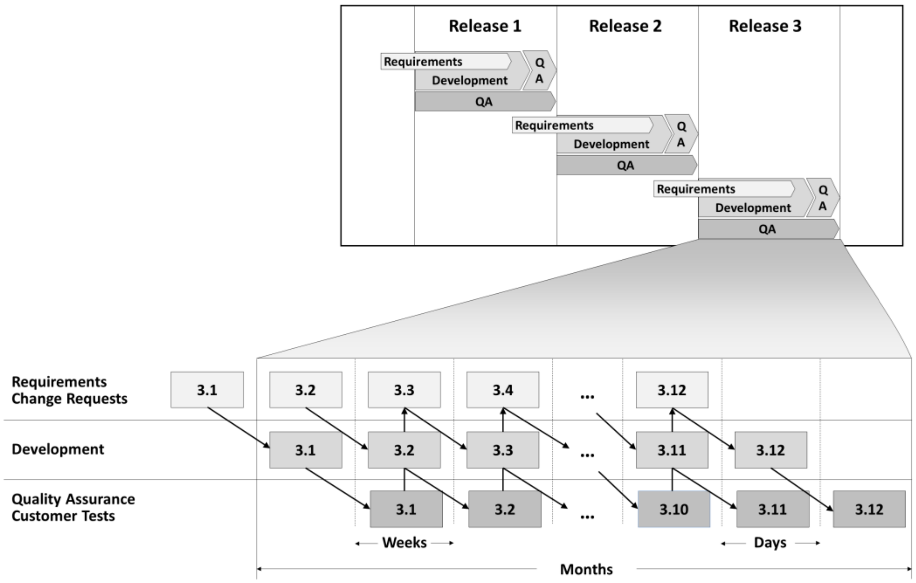

Fig. 1. Mgm's "Very Early Testing" quality assurance process

For each input item, i.e. a field which, as explained above, may appear in several instances, values need to be generated. The latter must comprise as many predefined extreme and special values (ESVs) as possible. Each ESV should, if feasible at all, occur in at least one test data record. There is no requirement to consider many (all???) combinations of interesting values, as a "combinatorial explosion" would arise. If a test requires a particular combination of values, then these values are predefined for the test data generation process, or such test cases are executed manually.

The execution of functional test cases (each using a single record from the test data set) is time consuming, since each test may run for several minutes. Therefore the size of a set with sufficient test coverage, as defined above, should be minimized. This requirement entails that the ESV density in each test data record should be maximized.

Within a software development cycle the rule-base associated with an application usually undergoes several changes. If such changes are small, a large proportion of existing test data is usually still valid, and can therefore be used for testing the application. But if the rule-base changes are substantial, new test data will have to be generated. This requirement entails that the test data generation process has to be correspondingly fast. For reasons of practicality our goal has been to accomplish a turn-around time of one day between the delivery of a new rule-base and the end of the data generation process.

For the test of large applications it is impossible to *manually* generate a sufficient amount of test data records with the required quality and within such stringent time limits. Therefore an *automated* generation approach is mandatory.

3 An Automated Test-Data Generator

The automation of test data generation – known to be a challenging, highly complex task – is not a complete novelty. Early trials, which date back almost four decades, even include the treatment of systems of non-linear equations [2]. Test data generators described in the literature (see e.g. [3]) are often based on a mathematical modeling process comprising the following phases: (1) construct a control flow graph (usually by some form of code analysis), (2) select an execution path, and (3) generate test data for the latter. Each execution path entails a so-called "path predicate". If the predicate is sufficiently simple, it can directly be submitted to a suitable solver (see e.g. [4], [5]). If there is no solution, the path cannot be followed at run-time.

All these methods have to satisfy some constraint(s), but since solving constraint satisfaction problems (CSPs) is particularly difficult, often one has to resort to heuristic techniques [3].

Over the past five years, the quality assurance division at mgm has developed an automated rule-based test data generator (R-TDG) fulfilling the requirements described above. Our test data generator resembles the classical method of generating test data insofar as it also uses predicates and a CSP-solver. However, unlike the classical method, we do not have to perform any code analysis (which is notoriously difficult) in order to generate a predicate. Instead, we are exploiting the validation rule-base that accompanies each of our form-centric applications. From the rule-base a "fundamental predicate" is directly derived, which concisely describes the domain of valid input data. This predicate is systematically varied in order to incorporate all those ESVs that should be present in a comprehensive, valuable set of test data records. Each variation represents a CSP that, probably after some simplifications, can directly be submitted to a CSP-solver, a Satisfiability Modulo Theories (SMT) solver in our case.

In principle, the operating procedure for the R-TDG is straightforward:

1. Define several configuration parameters, including the number of desired instances of the forms and of the field lines.
2. Define required ESVs for all field instances.
3. Add definitions representing new data types and functions.
4. Run the R-TDG and produce random test data records.
5. Validate these records, and feed them into functional tests.

Figure 2 shows how this procedure integrates with the development process of the application and its corresponding rule set. Three triggers may initiate a test data generation process:

1. A major release of a rule set containing new field definitions or constraints: Test data have to be generated for a functional test of the application.
2. A minor release of a rule set containing updated rules and only minor changes in field definitions: Old test data may be reused, or new test data have to be generated for a functional test of the application.
3. The deployment of a new version of the application for a functional test: New ESVs may be necessary, requiring the generation of new test data for a functional test of the application.

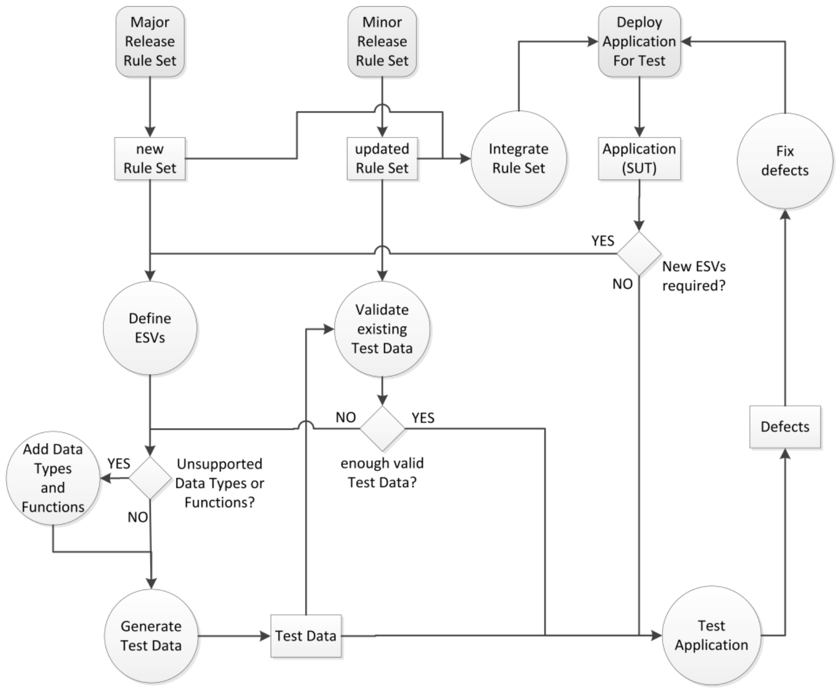

Fig. 2. Procedure for testing using a new or updated rule set, or a new version of the application. The necessary test data are generated by the R-TDG.

The data-generation procedure within the R-TDG works as follows:

1. Read the configuration.
2. Read the field and rule definitions
3. Translate each field definition into a number of associated variable definitions, and each validation rule into a number of associated constraints.
4. Assemble variables and constraints into a base CSP.
5. Read the field definitions, and generate ESVs for each field.
6. Use ESVs for creating variants of the base CSP.
7. Solve the resulting CSPs using an SMT-solver.
8. Translate the solutions back into the format required by the application.
9. Validate these records.

However, the simplicity of this process is misleading since a number of problems arise which have to be solved before valid test data emerge. Let us discuss these problems in some more detail.

3.1 Translating Field and Rule Definitions into Variable and Constraint Definitions

Several configuration parameters govern the whole data generation process. Two of those influence the translation, namely the actual number of instances of the forms holding the fields (e.g. instances for up to 14 children in a tax declaration), and the actual number of field instances. We refer to the actual numbers of form and field instances as their "multiplicities".

Consider, for example, the declaration of travel expenses: one "line" (consisting of a set of field instances) might consist of the amount of trips, the origin (i.e. the address of your employer), the destination (i.e. address of your destination), and the distance. A validation rule might restrict the number of trips to 366. When we generate test data, we have to set the desired multiplicity to less than 366 for the fields above.

A form multiplicity of m combined with a field multiplicity of n leads to m times n input items. The multiplicity values m and n can be quite large (1000 and more), but in practice we rarely use values exceeding 2.

An important obstacle to deal with consists in the fact that an input item may be empty. (An item being empty is equivalent to a Java-variable containing the value null.) No SMT-solver known to us can handle variables that have no value. Therefore, for each item, we have introduced a binary "occupation variable" that holds the information whether the corresponding "value variable" is null or not. Such a variable pair is semantically equivalent to a single variable whose value may be unspecified. For instance, a field item of type Boolean is translated into a variable pair that together can represent the three values: *true*, *false*, and *undefined*. We therefore refer to the logic implemented in the rules as "three-valued" logic.

Fields are declared in field definitions which, apart from the data type, hold additional information such as the minimum/maximum field length (number of characters). This information has to be translated into variable and constraint definitions that are comprehensible to the SMT-solver. Each field acts as a template, which is translated into $2 * m * n$ corresponding variables.

The rules, which form the other half of the input to the R-TDG, are also mere templates since initially only the *maximum* number of instances of the forms and of the fields occurring in those rules are specified. Therefore each abstract rule must be replicated according to the *actual* multiplicity of the forms and fields occurring in that rule. As a consequence, each abstract rule is translated into several concrete constraints. The constraints have to be formulated in such a way that they properly accommodate the value and occupation variables explained above. In effect, each rule is normally translated into $m * n$ corresponding constraints.

Another important issue to deal with is functions that occur in rule conditions. SMT-solvers do not comprehend proper functions, but only constraints over a very limited set of data types. Therefore each function occurring in a rule condition needs to be translated into a corresponding equivalent constraint.

The three-valued logic vastly complicates all logical expressions and all functions operating on field items. Consider the simple predicate $z = \max_2(x, y)$. This expands to 4 cases:

1. $z = \max(x,y)$, if both x and y have values (here max is the ordinary maximum function, which can be resolved into a logical condition),
2. $z = x$, if x has a value and y does not,
3. $z = y$, if y has a value and x does not,
4. $z = lb$, if neither x nor y is defined (here lb is the lower bound for the variables x and y).

A concise representation of some functions is also an issue. Consider, for example, a predicate y = f(x_1, x_2, …, x_n) which requires that at least one of the number of input items x_1, x_2, …, x_n has to have a non-null value. If for the occupation variables we were using ordinary Boolean data types (taking the values *true* or *false*), a complex and long winding expression for the function f would result. If we instead use bit data types (taking the values 0 or 1), a very compact formulation of the predicate emerges in form of a numeric inequality.

3.2 Representing Data Types Unsupported by SMT-Solvers

The translation process described so far is incomplete. What is missing is a description of how to deal with data types which occur in the rule-bases, but which have no direct counterpart in the SMT-world. A case in point is a decimal number which is an important data type for form-centric applications. Other such data types encountered in our rule-bases are calendar dates and date ranges. Again, no SMT-solver known to us can directly deal with calendar dates. We finally mention the important string data types. While dealing with string data types and string constraints is an important current research topic, none of the off-the-shelf SMT-solvers is capable of dealing with strings.

Below we discuss in some detail how we represent these data types in our CSPs.

Decimal Numbers

A decimal number is a rational number that possesses a representation with a finite number of digits after the decimal separator. For instance, a currency amount is a decimal number with at most two decimal digits. On the other hand the rational number 1/3 is not a decimal number.

In order to represent a decimal number within a CSP we use a pair consisting of a rational number and an accompanying constraint (called "decimal constraint"). The latter requires that a certain multiple of the number must be an integer. E.g. a standard currency amount with 2 post-decimal digits (say Euros with Cents) multiplied by 100 must be an integer.

Unfortunately those innocent looking decimal constraints can lead to severe performance issues for the SMT-solver.

Calendar Dates

Fields with values of type calendar date present another problem since the available SMT-solvers do not encompass the data type "calendar date". In the rule-base the (external) representation of a date value is always a string, such as "11.03.1956", but a typical cross field constraint uses functions that require separate access to the day in the month, the month, and the year of the calendar date. Comparisons with other date variables or constants, such as *before*, *at the same time*, or *later*, may have to be performed on parts of a date value, or on the whole value.

In order to enable a CSP-solver to operate on calendar dates, we represent any calendar date by an integer equivalent to a 'relative' day, starting from 01.01.1900 (which is day 1). In an imperative or functional programming language it is easy to implement accessor functions that retrieve the year, the month in the year, or the day in the month from such a relative day. However, we are dealing with a *declarative* CSP-language, and thus these accessors have to be implemented as constraints – a non-trivial task.

We found representations for all required date constraints. However the run-times are sometimes prohibitive. The only solution we have found so far consists in pre-assigning suitable values to a sufficient number of the date variables involved, and thereby remove these variables from the CSP in question. With this drastic measure we have been able to cut down the run-times to reasonable values, at the expense of losing some test-coverage.

Strings and String-Constraints

In a typical form-centric application a large number of the fields are string fields. Each such field can contain a maximum number of characters, which is a simple single-field constraint. The character set that the user may choose from for his/her input is another constraint. In our applications, many fields are further constrained by one or more regular expressions.

Here is a simple example of the combination of two regular expressions due to different rules: an ID code must match the expression "[0-9]{5}", but not match "00000", which might be a pseudo-value reserved for "first time customer who has no ID yet".

Solving CSPs over string variables is a current research topic (see e.g. [6], [7], and references therein). However, none of the off-the-shelf SMT-solvers presently includes a string data type.

In order to cope with string fields and associated string constraints we have used a heuristic approach that consists of the following elements: for a given field, the field-length constraint, the alphabet (character set) constraint, and any regular expression match constraint are considered as predicates over the field. Each predicate is replaced by the Boolean abstraction of the predicate, i.e. a Boolean variable which holds the logical value of the predicate. The modified CSP is then submitted to the SMT-solver, and the values of the artificial Boolean variables are read off the solution.

For each string field, the solution of the CSP consists of a list of regular expressions along with a list of Boolean values (match flags), which indicate whether the string value for the field should match the expression or not. In order to generate valid strings, fulfilling these predicates, a string generator was developed based on a publically available regular expression package [8]. Our string generator uses the well-known representation of a regular expression as a finite state automaton (see e.g. [9]). By walking the graph representing the automaton, strings can be generated that, in addition to meeting all constraints, may attempt to fulfill further requirements. The most important requirement is to exhaust the given character set as early and as well as possible. Our string generator accomplishes this goal.

The heuristic described above is not an exact method powerful to handle all occurring situations. Once more, the price to pay for our approximation consists in losing some of the viable solutions, and sometimes even generating inconsistencies. Fortunately, most of the time our heuristic works well and produces valid solutions to string constraints.

There are string constraints which, out of principle, cannot be treated by a string generator acting in a post-processing phase. These comprise the equality and inequality constraints between string variables, the substring function, and conversion functions where a suitable string is converted to a number or calendar date. These constraints do actually occur in our rule-bases, and properly solving CSPs that contain them would require a genuine string solver. In cooperation with the Technical University Munich such a string solver has been developed [7], but so far has not yet been incorporated into the productive R-TDG. Therefore in each of those cases a handcrafted workaround is still required.

3.3 Dealing with Non-linear Constraints

A major obstacle for almost any SMT-solver is non-linear constraints over numeric variables, such as $z = x * y$ for some variables x, y, and z (for an early account see e.g. [2]). Only recently a few SMT-solvers that can solve constraints comprising multinomial expressions have become available (see e.g. [10]).

For the time being, we linearize constraints such as the ones above, by manually replacing a sufficient number of variables by constant values. Of course, this way we lose ESV-coverage, but that is a modest price to pay, until we will be ready to move on to a solver capable of handling non-linear constraints.

3.4 Generating Extreme and Special Values

The R-TDG is expected not to produce arbitrary data records, but test data records that put the software under test (SUT) under pressure. Therefore, as explained above, the records have to include ESVs for the input items.

For a numeric input item an important ESV obviously is 0; other ESVs of interest are the minimum and the maximum admissible values. For a calendar date field 28th and 29th of February, and 1st of March are interesting ESVs. Dates at the boundaries of a quarter such as 1st of January and 31st of March are also interesting values. For a string field, the empty string and a string with the maximum allowed number of characters are interesting ESVs. For string fields it is important that each admissible character occurs in at least one ESV, if feasible at all. Of course, for all input items the null value is an important ESV.

For the production of ESVs we have implemented an automated ESV-generator. For each variable it produces a reasonable set of ESVs on the basis of the variable's generic field-type combined with some additional field-specific parameters such as the minimum/maximum field length.

In addition to predefined ESVs we usually include some random values that are treated in the same way as the deterministic ESVs.

The number of ESVs for a given CSPs is roughly proportional to the number of variables present. The number of ESVs to be generated for a given CSP averages about 3 to 5 times the number of variables contained in the CSP.

4 Solving the CSPs Efficiently

The problem of validating a given data set is straightforwardly solved with an algorithm with polynomial computational complexity. However, the associated *inverse* problem of generating test data has a computational complexity that is much higher than that of the *forward* problem. In almost all cases the problem of solving a CSP is NP-complete. It is therefore often difficult to obtain solutions to practical CSPs which, as in our case, may have many thousands of variables and many thousands of constraints.

Usually there no stringent correlation between the run-time and the size of the CSP measured by the number of variables or constraints. Nevertheless, CSPs with roughly the same number of variables and of constraints tend to be more difficult to solve than CSPs with an unbalanced number of variables and constraints. With few constraints compared to the number of variables the solution space is large, and with many more constraints than variables the search space for solutions can usually quickly be restricted, or contradictions are found which lead to an empty solution space.

Efficiency-boosting techniques are essential for a test data generator that is supposed to be useful in practice. It is important to offer reasonable turn-around times that fit to operational schedules. With a combination of measures we were able to reduce the make-span for generating tests data, even for our largest form-centric applications from about a week to about an hour. Some of those measures are explained below.

4.1 Partitioning a CSP into Independent Components

A major breakthrough consisted in partitioning any given CSP into independent CSP-components. Two variables are in the same component when they simultaneously occur only in constraints of this component. The components of a CSP can best be viewed in the undirected *primal constraint graph* in which each vertex corresponds to a variable, and two vertices are linked when the corresponding variables appear together in one of the constraints. Each component-CSP can be solved independently of the others.

One might think that CSP partitioning would be an integral part of any SMT-solver, but apparently that seems not to be the case. Fortunately, our external CSP-partitioning technique offers some advantages:

1. The partial solutions to the component-CSPs can freely be combined to form complete solutions to the parent CSP, i.e. valid test data records.
2. A trivial parallelization (i.e. concurrency) of the solution process becomes feasible, which further reduces the make-span for the test data generation. We are usually employing two to four threads, each operating on a different CSP-component. This approach exploits the resources of a state-of-the-art CPU with four physical (eight virtual) cores quite well. The reduction of the make-span is roughly proportional to the number of threads employed.

4.2 Decomposing a CSP via Cluster Analysis

The run-time of solving a given CSP is usually dominated by the run-time of solving the largest component-CSPs. Sometimes the largest component-CSPs still are too complex for our SMT-solver. Here it would be helpful if we could somehow identify one or more central variables which, if "removed" from the CSP, would permit the latter to be decomposed into two or more independent components. Again the child CSPs should more easily be solvable than the parent CSP.

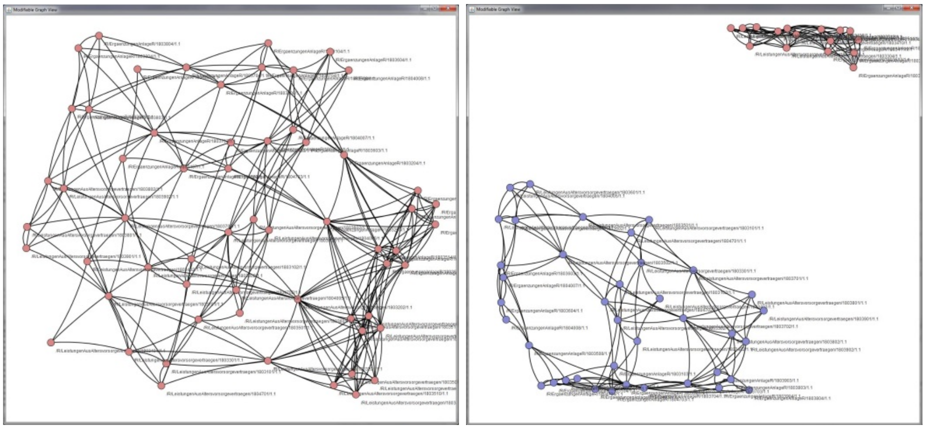

Fig. 3. CSP-decomposition accomplished by the "vertex inbetweenness" cluster analysis algorithm. The primal constraint graph for a fairly large component of the CSP corresponding to SUT #1 (see table 1) is shown before (left) and after (right) running the algorithm. The algorithm has removed four central vertices in order to accomplish the decomposition.

In order to tackle this problem we have developed a very fast, concurrent, graph-based "vertex inbetweenness" cluster algorithm that iteratively identifies central graph vertices. The algorithm works on the primal constraint graph of the problematic component-CSP. A central vertex, once identified, can subsequently be removed from the graph (figure 3). After we had developed our algorithm we found that it had already been invented several decades before in the field of sociology [11].

In the context of CSPs a variable removal can be achieved by pre-assigning a value to the corresponding variable. (The pre-assigned value effectively transforms the variable into a constant which therefore disappears from the CSP.) Often the removal of a single variable does not yet allow a decomposition of the CSP. If so, the cluster algorithm has to be iterated until decomposition becomes possible.

The vertex-inbetweenness algorithm is really amazing in identifying the important central vertices, which eventually will allow a decomposition of the CSP into independent CSP-components of roughly comparable size. (The algorithm is greatly superior to e.g. the simple vertex-cut algorithm, which will often split a given CSP into one large and one tiny component.) From observing the effect of the vertex-inbetweenness algorithm, and from analyzing its inner workings, we can state that it behaves as if it were “goal directed”. It seems to identify a potentially worthy cut consisting of one or more central vertices. During each iteration one of the vertices in the cut-set is removed until the cut has been achieved. Only after having accomplished this task the algorithm considers another potential cut.

The cluster algorithm is *the* power tool in our toolbox which we apply when nothing else helps to reduce the run-time of a critical (i.e. prohibitively long-running) component-CSP. We have had situations where the SMT-solver, working on a relatively small component-CSP, would not return within half a day. This, of course, is unacceptable. The CSP-decomposition, accomplished by applying the cluster algorithm, usually helps to dramatically reduce the make-span which the SMT-solver requires for generating solutions – sometimes by several orders of magnitude. The performance gain again comes at the expense of test coverage, since variable values oftentimes are fixed to constants that may not even be ESVs. However, obtaining test data with reduced test coverage is much better than obtaining test data too late or not at all!

4.3 Re-using Partial Solutions

Another very important efficiency boosting technique, which we recently implemented, consists in the following: when, for a component-CSP which is currently being worked on, a partial solution containing some ESVs has been obtained, it can be reused as a starter, when other ESVs are being added. For large rule-bases, reusing partial solutions has decreased the make-span for obtaining complete test data sets by a factor of 30 to 50. To us this large reduction factor came as a welcome surprise.

4.4 Handling Decimal Constraints

At first sight decimal constrains appear innocent. However, in practice we all too often suffer from severe efficiency problems. Particularly when the number of post-decimal digits is variable (as is the case for some fields in our SUTs), the run-time for solving the CSPs can increase dramatically. We therefore had to resort to the following heuristic: we solve the CSP without decimal constraints, and wait for a problem with decimal numbers to surface in at least one of the solutions. Only for those problematic variables, where a solution contains a rational number that cannot be represented as a decimal number with the allowed number of post-decimal digits, a decimal constraint is inserted into the CSP. The CSP is then solved once more.

Clearly, this heuristic approach requires two or more solution runs. However, in our experience, the overall efficiency, when using this heuristic, is still a lot higher compared to inserting a decimal constraint for all variables that represent a decimal number.

4.5 Timeouts

Sometimes the solution process for a component-CSP is well underway, when all of the sudden the SMT-solver “goes on strike”, meaning, it does not return within acceptable time. In such a situation a timeout is very helpful. When a process, in which an SMT-solver runs, is overdue, it is cancelled. The latest ESV added to the current component-CSP is considered to be the trouble-maker, and is eliminated from further consideration. The solution process is then restarted, and it is guaranteed that the process not only terminates, but terminates in acceptable time.

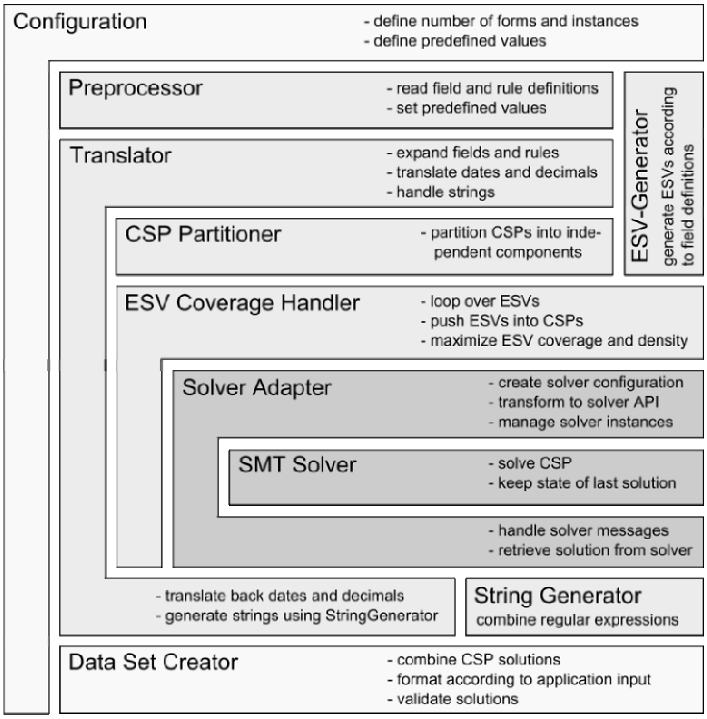
Configuration
- define number of forms and instances
- define predefined values
Preprocessor
- read field and rule definitions
- set predefined values
Translator
- expand fields and rules
- translate dates and decimals
- handle strings
ESV-Generator
generate ESVs according to field definitions
CSP Partitioner
- partition CSPs into independent components
ESV Coverage Handler
- loop over ESVs
- push ESVs into CSPs
- maximize ESV coverage and density
Solver Adapter
- create solver configuration
- transform to solver API
- manage solver instances
SMT Solver
- solve CSP
- keep state of last solution
- handle solver messages
- retrieve solution from solver
- translate back dates and decimals
- generate strings using StringGenerator
String Generator
combine regular expressions
Data Set Creator
- combine CSP solutions
- format according to application input
- validate solutions

Fig. 4. Architecture of the rule-based test data generator (R-TDG)

5 Architecture

A sketch of the architecture of the R-TDG is shown in figure 4. The basic idea is to start with a rule set, translate it to a CSP with variables and constraints, partition the CSP into independent components, solve the CSP by an SMT-solver (which is accessed via a “Solver Adapter”), assemble the solutions of the CSPs, and translate them back to valid test data records. Each architectural component contained in the diagram is accompanied by a short description of its task.

The most complex components are the “Translator” and the “ESV Coverage Handler”. The Translator remedies the discrepancy between the “real world” definitions of fields and rules of the business domain and the available variable types and constraint definitions of an SMT-solver. The ESV Coverage Handler tries to include as many ESVs as are feasible into the resulting test data set, while keeping the overall size of the set at a minimum.

6 Results

Table 1 presents the results of generating test data for some large form-centric SUTs. For all SUTs the number of test data records is relatively small considering the number of ESVs that have to be incorporated. In addition, in all cases the make-span of generating test data is between ~10 and ~100 minutes, well below the one day target that we set out, when we began the development of the R-TDG.

Table 1. For several large form-centric software applications under test (SUTs) the table shows the size of the CSP (measured by the number of variables and the number of constraints), the number of CSP-components after partitioning and decomposition, the number of test data records produced, and the make-span of the data generation. Form and field multiplicities were set to two. Two threads were used in parallel.

SUT	# Variables	# Constraints	# Components	# Records	Make-span
#1	11,845	14,307	3309	51	25 min
#2	13,325	17,063	2129	90	92 min
#3	3,128	4,111	374	79	9 min
#4	2,934	3,736	381	86	12 min
#5	4,830	5,968	712	86	11 min
#6	4,199	5,069	452	79	19 min

7 Summary

We have presented a novel approach for generating artificial random test data that are suitable for testing large form-centric software applications. These contain up to several thousand fields that the user has to potentially fill in. The very same single-field and cross-field constraints that are used by a validator, for validating the user input to the application, are also used by our test data generator.

The set of base constraints is augmented by simple additional constraints, which insert into the solutions "extreme and special values" (ESVs), whose purpose is to exert pressure onto the software application under test (SUT). The variations of the initial constraint satisfaction problem (CSP) are solved by an off-the-shelf Satisfiability Modulo Theories (SMT) solver. Data types unknown to current SMT-solvers such as decimal numbers, calendar dates, and strings, have to be treated in special ways. Several heuristics, including an effective graph-clustering algorithm, have been put in place in order to enable an efficient generation of test data. Even for large form-centric applications the make-span for data generation has come down to less than 100 minutes.

For about four years, these data are regularly used mainly for automated tests of several large form-centric software applications that are being developed by our company.

8 Glossary

CSP	constraint satisfaction problem
ESV	extreme and special value
R-TDG	rule-based test data generator
SMT	satisfiability modulo theories
SUT	software (application) under test

References

1. Dost, J., Nägele, R.: “jFunk Overview”, mgm technology partners GmbH (2012)
2. Howden, W.E.: Methodology for the Generation of Program Test Data. IEEE Transactions on Computers C-24(5), 554–560 (1975)
3. Edvardsson, J.: Survey on Automatic Test Data Generation. In: Second Conf. on Computer Science and Engineering in Linkoeping (ECSEL), pp. 21–28 (1999)
4. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), 900–910 (1991)
5. Gotlieb, A., Botella, B., et al.: Automatic test data generation using constraint solving techniques. ACM SIGSOFT Software Engineering Notes 23(2), 53–62 (1998)
6. Hooimeijer, P., Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. Redmond City, Microsoft Research (2010)
7. Braun, M.: A Solver for a Theory of Strings. Fakultät für Informatik, Technische Universität München (2012)
8. Møller, A.: “Automaton.” Aarhus, Basic Research in Computer Science (BRICS) (2009)
9. Brüggemann-Klein, A.: Regular expressions into finite automata. In: Simon, I. (ed.) LATIN 1992. LNCS, vol. 583, pp. 87–98. Springer, Heidelberg (1992)
10. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver (2012)
11. Freeman, L.C.: A Set of Measures of Centrality Based on Betweenness. Sociometry 40, 35–41 (1977)