Towards Constraint Logic Programming over Strings for Test Data Generation

Sebastian Krings[1], Joshua Schmidt[2], Patrick Skowronek[3], Jannik Dunkelau[2], and Dierk Ehmke[3]

[1] Niederrhein University of Applied Sciences
Mönchengladbach, Germany
sebastian.krings@hs-niederrhein.de
[2] Institut für Informatik, Heinrich-Heine-Universität
Düsseldorf, Germany
[3] periplus instruments GmbH & Co. KG
Darmstadt, Germany

Abstract. In order to properly test software, test data of a certain quality is needed. However, useful test data is often unavailable: Existing or hand-crafted data might not be diverse enough to enable desired test cases. Furthermore, using production data might be prohibited due to security or privacy concerns or other regulations. At the same time, existing tools for test data generation are often limited.
In this paper, we evaluate to what extent constraint logic programming can be used to generate test data, focusing on strings in particular. To do so, we introduce a prototypical CLP solver over string constraints. As case studies, we use it to generate valid IBAN numbers, calendar dates and specific data in JSON.

1 Introduction

Gaining test data for software tests is notoriously hard. Typical limitations include lack of properly formulated requirements or the combinatorial blowup causing an impractically large amount of test cases needed to cover the system under test (SUT). When testing applications such as data warehouses, difficulties stem from the amount and quality of test data available and the volume of data needed for realistic testing scenarios [9]. Artificial test data might not be diverse enough to enable desired test cases [15], whereas the use of real data might be prohibited due to security or privacy concerns or other regulations [18], e.g., the ISO/IEC 27001 [17]. Further challenges have been identified by Khan and ElMadi [20].

In consequence, to properly test applications one often has to resort to artificial test data generation [18]. However, existing tools are limited as they

- generate data that does not cover the desired scenarios [15],
- are specialized and lack options for configuration and adaptation [16], and
- generate an amount of data that is unrealistic for the SUT [29].

In this paper, we evaluate to what extent constraint logic programming could be used for test data generation, in particular for generating strings. We are not concerned with software testing itself.

2 Test Data

The International Software Testing Qualifications Board (ISTQB) describes test data as data created or selected to satisfy the preconditions and inputs to execute one or more test cases [30]. Test data may belong to the following categories:

- status data, files or surrounding systems required for a reusable start state,
- input data transferred to a test object during test execution,
- output data returned by a test object after execution,
- production data, which is deducted from the production system.

Production data is often used for testing as it provides obvious test cases and can be gathered easily. However, using production data does not lead to thorough testing, e.g., it never contains dates in the future. While production data can be anonymized, it is hard to guarantee that de-anonymization is impossible. Furthermore, production data may be biased.

Those problems can be solved by generating synthetic data. The implementation of a test data generator for each specific problem is cumbersome. One just wants to describe the problem at hand without implementing the actual data generation. We therefore consider constraint programming to be appropriate for implementing general test data generators. In particular, relying on constraint programming provides a number of further benefits common to declarative languages: specification of data and generating programs are more closely related and maintainability is increased. Furthermore, constraint-based and logic programming allows to easily extend given specifications by further constraints and thus increases extensibility and combinability.

However, generating synthetic data remains a complex task as it involves thoroughly specifying constraints the data needs to fulfill in order to derive high quality test data.

2.1 Test Data Generators

The generation of synthetic test data can be supported by different test data generators [30]: Database-based generators synthesize data according to database schemata or create partial copies of database contents, i.e., they rely on production data. Interface-based generators analyze the test object's API and determine the definition areas of input parameters to derive test data from. In this context, test oracles cannot be derived.

Code-based generators take the source code of the SUT into account, which has disadvantages. For instance, it prevents oracle generation and is unable to work with source code that is not available (e.g., for foreign libraries). Furthermore, code-based generators are a weak test base, especially lacking the intellectual redundancy necessary for testing (four-eyes principle) [27], i.e., the understanding of how a system is supposed to work and how it is implemented are necessarily identical if tests are generated purely based on code.

Specification-based generators generate test data and oracles based on specifications written in a formal notation. A specification based generator could

thus generate data that replaces production data. The quality of the test data is ensured by the model and the correctness of the solver. This includes quality aspects such as conformity and accuracy. To build such a generator, constraint solving over all needed data types is required.

2.2 Requirements Towards Solvers

To gain a sensible set of requirements for a string constraint solver for test data generation, we decided to look at the feature set of Oracle SQL. The reasoning behind this is as follows: SQL was designed for the description of complex data flows and is therefore suited as a modeling language for test data generation [22]. It is widely used by developers, test data specialists and technical testers, i.e., they would be able to use it as a possible input language for generation tools. Additionally, SQL statements can easily be extracted from source code and can thus be used to automatically generate test data for given applications. Furthermore, SQL is declarative and offers a good level of abstraction.

There are several types of strings in Oracle SQL[4], in particular, unbounded unicode strings. In addition, other data types are required for practical test data: integers, fixed point numbers, reals and dates. There are no booleans in SQL, however, booleans ease encoding complex SQL conditions into constraints. Regarding BLOBs (e.g., images stored inside the database), we are so far not interested in supporting them, since SQL does not provide operations on them and their semantics are usually invisible to the applications.

Oracle SQL lists 54 functions on strings[5]. The ones we are interested in are

- CONCAT: concatenation of strings,
- LENGTH: returns the length of a string,
- REGEXP: tests, whether a string matches a given regular expression or not,
- SUBSTRING: returns a substring with given start position and length,
- TO_NUMBER: convert a string to number and vice versa.

Other operations can often be implemented with these functions or are not of interest for test data generation. REGEXP requires the solver to process regular expressions. The constraint handlers for all types must interwork, since dependencies can exist between variables of different types. While we expect correctness, we cannot expect (refutation) completeness, since once all desired operations are added the problem becomes undecidable [7].

3 Related Work & Alternative Approaches

In the following, we will briefly present alternative approaches to constraint logic programming. For a selection of alternative solvers, we will discuss their implementation paradigms, in order to later compare to constraint logic programming.

[4] `https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/tables-and-table-clusters.html#GUID-A8F3420D-093C-449F-87E4-6C3DDFA8BCFF`

[5] `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Functions.html#GUID-D079EFD3-C683-441F-977E-2C9503089982`

3.1 autogen

autogen [10] is a specification-based test data generator. autogen is able to directly use SQL as an input language. In order to generate test data from it, SQL is considered as specification of the SUT and is converted into constraints. autogen uses an independently developed string constraint solver called CLPQS, which handles all requirements stated in Section 2.2. To support the data types of SQL, autogen interacts with a set of different solvers. In particular, it relies on CLP(Q) and CLP(R) for rationals and reals, which have some limitations when it comes to completeness. CLPQS represents domains as regular expressions. One motivation for this paper is to experiment with different representations and propagation algorithms.

3.2 MiniZinc

MiniZinc is a solver-independent modeling language for constraint satisfaction and optimization problems. A MiniZinc model is compiled into a FlatZinc instance which can be solved by a multitude of constraint solvers. An extension of the MiniZinc modeling language with string variables and a set of built-in constraints has been suggested by Amadini et al. [3]. String variables are defined as words over the alphabet of ASCII characters and have a fixed, bounded or unbounded length. Yet, strings are represented as bounded length arrays of integers when translating to FlatZinc. The MiniZinc model itself does allow strings of unbounded length though. MiniZinc enables optimization over constraints rather than just satisfiability and allows mixing constraints over different domains. However, there are no direct conversions from other types to strings.

3.3 SMT Solvers

SMT solvers such as CVC4 [5] and Z3 [8] have been used for test case generation in the context of programming languages [31]. Both solvers support constraints over strings and regular expressions and are able to handle operations such as concatenation, containment, replacement and constraining the length of strings.

In Z3's original string solver, strings are represented as sequences over bit-vectors. The solver itself is incomplete and relies on heuristics. In contrast, Z3-str [39] introduces strings as primitive types. Z3-str leverages the incremental solving approach of Z3 and can be combined with boolean and integer constraints. There have been several improvements of Z3-str in recent years [33,38,6].

CVC4's string solver [24,25] allows mixing constraints over strings and the integers. The authors present a set of algebraic techniques to solve constraints over unbounded strings, usable for arbitrary SMT solvers.

Another SMT solver for string constraints is Trau [1], which, in contrast to CVC4 and Z3, supports context-free membership queries and transducer constraints by using pushdown automata. Trau implements a Counter-Example Guided Abstraction Refinement (CEGAR) framework, computing over- and

under-approximations to improve performance. Key idea in Trau is a technique called flattening [2], leveraging that (un)satisfiability can be shown using witnesses of simple patterns expressable as finite automata.

3.4 Other Solvers

Kiezun et al. presented Hampi [21], a constraint solver over strings of fixed length featuring a set of built-in constraints. Hampi is able to reason over regular languages. String constraints are encoded in bit-vector logic which are then solved by the STP [13] bit-vector solver. At the expense of expressiveness, limiting the length of strings enables a more restricted encoding, increasing the performance by several orders of magnitude. However, a bit-vector encoding has a larger memory consumption than using finite automata.

G-Strings [4] is an extension of the Gecode constraint solver [28]. Both solvers accept strings of bounded but possibly unknown length. In contrast to Gecode, strings are not represented using integer arrays but as a restricted language of finite regular expressions. This prevents the static allocation of possibly large integer arrays and thus improves performance.

Fu et al. introduced Simple Linear String Equations (SISE) [12], a formalism for specifying constraints on strings of unbounded length, and presented the constraint solver Sushi using finite automata to represent domains.

3.5 Summary

In summary, several approaches have been suggested for string constraints. However, no single approach is able to satisfactorily handle the requirements posed for test data generators described in Section 2.2. A comparison of different solvers considering the features described in Section 2.2 is shown in Table 1. The requirement of a combined solver states that a direct conversion between strings and other types is provided. As ConString has been developed specifically for this problem domain, it naturally supports the most requirements.

4 Constraint Logic Programming Over Strings

We implement a constraint logic programming system for strings using Constraint Handling Rules (CHR) [11] on top of SWI-Prolog [37] called ConString. We use classic constraint propagation to reduce variable domains. The system supports strings of unbounded length and is coupled with CLP(FD), CLP(R) and CLP(B) to handle the integers, reals and booleans respectively. While not all SQL string operations are implemented yet, we plan to do so in the future. One goal is to employ different techniques than CLPQS to compare and possibly improve both solvers. We think CLP is adequate since there are many other solvers to build up upon and since it provides access to all solutions using backtracking.

In the following, we present our encoding of string domains and discuss its advantages and drawbacks, followed by the currently featured constraints, selected constraint handling rules and solver integrations.

Table 1. Features of constraint solvers. (✓) indicates partial support or workaround.

Solver	Strings			Combined Solver		
	Unbounded	Unicode	SQL Operations	Integer	Boolean	Real
CLPQS	✓	✓	✓	✓	✓	(✓)
MiniZinc	✗	✗	(✓)	✗	✗	✗
CVC4	✓	✗	✓	✓	✗	✗
Z3-str3	✓	✗	(✓)	✓	✗	✗
S3	✓	✗	✓	✓	✓	✗
Hampi	✗	✗	(✓)	✗	✗	✗
Sushi	✓	✓	✓	✗	✗	✗
G-Strings	✗	✗	(✓)	✗	✗	✗
Trau	✓	✗	✓	✓	✗	✗

4.1 Domain Definition

To fulfill the requirements posed in Section 2.2, we decided not to enforce a fixed length of strings and to use regular expressions as input. The employed alphabet consists of ASCII characters and some special characters like umlauts and accented characters. Dynamic character matching is possible by specifying ranges (e.g., `[0-9a-f]`), or by using the dot operator. We match a whitespace in regular expressions by `\s` while actual whitespace characters can be used to structure regular expressions without being part of the accepted language.

Further, we support the usual regular expression operators on characters, i.e., quantity operators (`*`, `+` and `?`) and the alternative choice operator (`|`). For convenience, our regular expressions offer more strict repetition definitions noted by `{n}` (exactly `n` times), `{m,n}` (`m` to `n` times) and `{m,+}` (at least `m` times).

4.2 Domain Representation

Since ConString is supposed to handle strings of unbounded length, we represent domains as finite automata as done by Golden et al. [14]. First, this allows for a concise specification of regular languages with low memory consumption. Second, finite automata support basic operations such as union, intersection, concatenation or iteration and are closed under each of these operations. In particular, we use non-deterministic finite automata with ϵ-transitions.

Since SWI-Prolog does not have a native library for handling finite automata, we encode them as a self-contained term `automaton_dom/4` consisting of a set of states, a transition relation as well as a set of initial and final states. The states are a coherent list of the integers $1 \dots n$, $n \in \mathbb{N}$. The transition relation is implemented as a list of triples containing a state s_1, a range of characters (might contain a single character only) and a target state s_2 reached after processing a character from the range of characters in s_1, e.g., `(0, a, 1)`.

We implement the common operations on finite automata used for regular languages as well as basic uninformed search algorithms used to label automata,

i.e., to find a word having an accepting run. The search is backtrackable providing access to an automaton's complete language.

Efficiency The chosen representation of finite automata has several drawbacks. We use lists to store states and transitions providing linear time concatenation and element access leading to a loss of performance, especially when labeling automata. It would be desirable to use a data structure such as hashsets, which provide amortized constant time performance for basic operations. However, such a data structure is currently unavailable in SWI-Prolog[6].

Another drawback is that we have to rename states when performing basic operations on automata. For instance, the concatenation $\mathcal{A}_1.\mathcal{A}_2$ is implemented by using the final states of $\mathcal{A}_2$ for the resulting automaton and adding an ϵ-transition from all final states of $\mathcal{A}_1$ to all initial states of $\mathcal{A}_2$. In order to avoid ambiguities, the states of $\mathcal{A}_2$ have to be renamed by shifting their identifier names by the number of states in $\mathcal{A}_1$. This renaming is one of the main issues for efficiency as it adds a linear time complexity component with respect to the size of the second automaton to all the basic operations.

4.3 Constraint Handling Rules

We use CHR on top of SWI-Prolog providing the constraint store and propagation unit to reduce variable domains. Moreover, CHR serves as user interface.

The CHR language is committed-choice, i.e., once a rule is applied it cannot be revoked by backtracking. Rules consist of three parts: a head, a guard and a body. A rule is triggered as soon as the head matches constraints in the constraint store. Guards allow imposing restrictions on rule execution. Finally, the body consists of Prolog predicates and CHR constraints. Predicates are called as usual while constraints are added to the constraint store, possibly triggering further propagation. All available constraints are propagated until the constraint store reaches a fix point. Solving fails if an empty string domain is discovered.

CHR provides three different kinds of rules: First, propagation rules of the form `head ==> guard | body`, where the body is called if the guard is true. The head constraints are kept. Simplification rules of the form `head <=> guard | body` update the constraint store by replacing the head constraints by those derived from the body. Simpagation rules of the form `head1 \ head2 <=> guard | body` are combined rules, retaining the constraints of the first part of the head while discarding those of the second part.

Our implementation currently supports several basic operations on regular languages such as intersection, concatenation or iteration as well as a membership constraint, arithmetic length constraints (fixed or upper bound), string to integer conversion, prefix, suffix and infix constraints and case sensitivity constraints. For now, we ensure arc- and path-consistency of our constraints. Variables can be labeled using `str_label(+Vars)` or `str_labeling(+Options,`

[6] While SWI-Prolog has built-in support for dictionaries, element access is logarithmic and updates are linear in size.

Listing 1. CHR rules for the membership constraint `str_in/2`.

```
1 str_in(S1, S2) <=>
2     string(S2) | gen_dom(S2, D), str_in(S1, D).
3 str_in(_, D) ==> is_empty(D) | fail.
4 str_in(Var,D) ==> D = string_dom(Cst) | Var = Cst.
5 str_in(S, D1), str_in(S, D2) <=>
6     D1 \= D2 | intersection(D1, D2, D3), str_in(S, D3).
7 str_in(S, D1)\ str_in(S, D2) <=>  D1 == D2 | true.
```

Listing 2. CHR rules for the concatenation constraint `str_concat/3`.

```
1 str_in(S1, D1), str_in(S2, D2), str_concat(S1, S2, S3) ==>
2     concat(D1, D2, D3), str_in(S3, D3).
3 str_in(S1, D1), str_concat(S1, S1, S3) ==>
4     concat(D1, D1, D3), str_in(S3, D3).
```

`+Vars)`. As options, we currently support selecting the search strategy for automata (`dfs`, `idfs`, `bfs`) and any option on integer domains provided by SWI-Prolog's CLP(FD) library. In the following, we will describe selected constraint handling rules in more detail.

The membership constraint is defined as shown in Listing 1. The first rule is applied in case membership is called with a string or regular expression. Then, a finite automaton representing the input domain is generated and the same constraint is applied to this automaton domain. The second rule states that whenever a domain is empty constraint solving should fail as no solution exists. Third, in case the string domain becomes constant, we propagate the value to the variable. The fourth rule joins two non-equal membership constraints for the same variable by intersecting both domains and replacing the two constraints by a single membership constraint. A final rule is used to remove one of two identical membership constraints. Note that `gen_dom/2` and `intersection/3` are called for internal domain computation and not added to the constraint store.

Concatenation is defined using two rules as shown in Listing 2. It relies on the membership constraint by assuming that two `str_in/2` refer to different variables. The first rule defines the concatenation of two different string variables by concatenating their automata domains and adding a new membership constraint for the result. Analogously, the second rule defines the concatenation of the same string variable onto itself. In order to efficiently propagate a constant string result to the first two arguments, we add a third rule using SWI-Prolog's string concatenation, e.g., `string_concat(A, B, ''test'')`, providing all solutions on backtracking. If a candidate has been found, it is checked upon labeling whether the candidate is accepted by the corresponding domains. If so, membership constraints are propagated assigning constant values to all arguments.

Listing 3. CHR rules for the infix constraint `str_infix/2`.

```
1 str_infix(S, IStr) <=>
2     string(IStr) | gen_dom(IStr, IDom), str_infix(S, IDom).
3 str_infix(S, IDom) <=>
4     any_char_dom(A), repeat(A, AStar),
5     concat(IDom, AStar, T), concat(AStar, T, ResDom),
6     str_in(S, ResDom).
```

The iteration operation `str_repeat/[2,3,4]` is defined as repeated concatenation. Case sensitivity operations are defined by setting up membership constraints to generated domains accepting only upper or lower case characters.

The infix operation `str_infix/2` for two string variables s_1 and s_2 is defined by adding a membership constraint for s_1 to be an element of the regular language $\mathcal{L}(.^*).\mathcal{L}(s_2).\mathcal{L}(.^*)$ as shown in Listing 3. Again, the first rule is a wrapper generating a finite automaton domain from a string or regular expression. Prefix and suffix operations are defined in the same manner.

4.4 Integration of CLP(FD), CLP(R) and CLP(B)

In order to enable the generation of richer test data and allow for a greater coverage of test scenarios, we extend ConString to support combining constraints over different domains. In particular, we support constraints over finite domain integers using CLP(FD) [34], constraints over reals using CLP(R) and constraints over booleans using CLP(B) [35,36]. As an interface, we provide the bidirectional constraints `str_to_int/2`, `str_to_real/2` and `str_to_bool/2`.

The implementation of `str_to_int/2` consists of four rules as shown in Listing 4. In order to detect failure early we check for inequality if both arguments are constants. If only the integer variable is a constant, we convert and assign the value to the string. In the third rule, a constant string is assigned to the integer variable. Note that `number_string/2` removes leading zeros by default. Besides that, we provide a rule to fail for constant strings not representing integers.

We additionally provide a second implementation `str_to_intl/2` allowing leading zeros in order for constraints such as `str_to_int("00", 0)` to hold. This is achieved by additionally concatenating the domain of 0^* to `IDom` in line 6 of Listing 4.

The integration of CLP(R) and CLP(B) is implemented analogously propagating membership constraints to a specific backend if variables are constant values. Again, alternative implementations are provided allowing an arbitrary amount of leading zeros when converting from string to boolean or real.

5 Case Studies

In this section, we will present three case studies of using ConString: a generation of IBANs, calendar dates, and data tables in JSON. Finally, we will conclude

Listing 4. Basic rules for the integration of CLP(FD) propagating constant values.

```
1  str_to_int(S,I) ==>
2    string(S), integer(I), number_string(SInt, S), I \== SInt |
3    fail.
4  str_to_int(S,I) ==>
5    integer(I), number_string(I, IString) |
6    cst_str_dom(IString, IDom), str_in(S, IDom).
7  str_to_int(S,I), str_in(S,D) ==>
8    D = string_dom(CstString),
9    number_string(CstInteger, CstString) | I #= CstInteger.
10 str_to_int(S,_), str_in(S,D) ==>
11   D = string_dom(CstString), \+ number_string(_, CstString) |
12   fail.
```

this section with a discussion of the benefits from constraint logic programming compared to typical test data generators.

5.1 Generation of IBAN Numbers

As a case study, we specify the computation of valid International Bank Account Numbers (IBANs) as a constraint system as done by Friske and Ehmke [10]. This example is of interest as it yields a relatively large search space and requires the conversion between the integers and strings. Generated data can, for instance, be used to initialize unit tests of components validating IBANs. This example is an excerpt of a project where an interface between a SEPA credit transfer and a micro-service managing financial push notifications has been tested.

A German IBAN consists of 22 characters which are characterized as follows: The first two characters represent the country code (here, the constant "DE") while the third and fourth characters are a checksum. The remaining 18 digits represent the Basic Bank Account Number (BBAN).

We can compute valid IBANs using a given country code as follows: Represent the country code as a digit where "A" equals 10, "B" equals 11, etc. The German country code "DE" is hence encoded as 1314. Concatenate two zeros to the encoded country code (i.e., 131400) and prepend the BBAN. This forms a 24 digit number, σ_b. In order to compute the valid checksum σ_c, the constraint $98 - (\sigma_b \; mod \; 97) = \sigma_c$ must hold. Finding a solution binds the BBAN to a value in its domain and provides its corresponding checksum σ_c. To derive the actual BBAN, remove the suffix "131400". Finally, concatenate the computed checksum σ_c with the BBAN and prepend the country code "DE" as a string.

The complete constraint system is shown in Listing 5. Lines 3 and 4 define the BBAN and the 24 digits number σ_b respectively. The constraint for computing σ_c is set in line 5. The remaining specification is straightforward as described above. Note that we allow leading zeros for the checksum's string.

Listing 5. Constraint system to compute all valid german IBANs.

```
iban(IBAN) :-
    SigmaC in 0..96,
    BBAN in 100000000000000000..999999999999999999,
    SigmaB #= BBAN * 1000000 + 131400,
    SigmaB mod 97 #= SigmaC,
    str_label([SigmaB, SigmaC]),
    str_to_int(BBANStr, BBAN),
    CheckSum #= 98 - SigmaC,
    str_to_intl(CheckSumStr, CheckSum),
    str_size(CheckSumStr, 2),
    str_in(DE, "DE"),
    str_concat(DE, CheckSumStr, IBANPrefix),
    str_concat(IBANPrefix, BBANStr, IBAN),
    str_label([IBAN]).
```

Table 2. Benchmarks for generating IBANs. Walltime in seconds.

Amount	1	10	100	1,000	10,000	100,000	250,000
CLPQS	0.006	0.024	0.240	2.029	32.163	1525.457	9261.204
ConString	0.007	0.038	0.105	1.066	26.573	1342.597	9841.225

For benchmarking, we generate sets of IBANs of varying sizes, using an Intel Core i7-6700K with 16GiB RAM. We used SWI-Prolog's predicate `statistics/2` to measure the walltime. Table 2 shows the median time of five independent runs and compares our solver with CLPQS. As can be seen, ConString performs overall slightly better than CLPQS with the exception of the generation of 250,000 IBANs. Up to one thousand samples both solvers appear to scale linearly. Notable exception is the jump from 10,000 to 100,000 generated samples. Here, both solvers scale worse: CLPQS scales with a factor of 47, whereas ConString takes 50 times as long as for generating 10,000 IBANs instead of the expected factor of 10. At least for ConString, experimental results have shown that this non-linear growth is caused by SWI-Prolog's CLP(FD) library.

We also encoded the example in SMT-LIB to compare ConString and CLPQS with Z3-str3 and CVC4. Unfortunately, CVC4 did not return a result but timed out after 600 seconds. Z3-str3 found a single solution in around 0.2 seconds. We were unable to compute multiple solutions using Z3-str3 as the solver timed out searching for further ones.

5.2 Generation of Calendar Dates

Another example is the generation of various date expressions, which is of interest for testing for many tools which need to parse valid dates and reject invalid

Listing 6. Constraint system to compute diverse calendar date expressions.

```
1  date(Date) :-
2    WeekDay str_in "Monday|Tuesday|...|Sunday",
3    Month str_in "January|February|...|December",
4    Day str_in "[1-9]|[1-2][0-9]|3[0-1]",
5    Year str_in "[1-9][0-9]{0,3}",
6    MonthDay match Month + "_" + Day,
7    MonthDayYear match MonthDay + ",_" + Year \/ MonthDay,
8    FullDate match WeekDay + ",_" + MonthDayYear,
9    Date match MonthDayYear \/ FullDate \/ WeekDay,
10   str_label([Date]).
```

Table 3. Benchmarks for generating date expressions. Walltime in seconds.

Amount	1	10	100	1,000	10,000	100,000
CLPQS	0.000	0.000	0.000	0.000	0.000	0.010
ConString	0.010	0.010	0.010	0.011	0.080	0.965

ones. The accepted expressions are of either of the forms "Tuesday", "August 30", "Tuesday, August 30", "August 30, 2016", or "Tuesday, August 30, 2016".

Listing 6 shows the corresponding constraints, taken from [19, Section 3]. The constraint system consists of defining the basic building blocks first: the weekdays, the months, and valid year numbers. Thus, only the years 1 to 9999 are accepted. Further, the more complex parts are constructed, each consisting of a combination of operations on variables constrained before. This leads up to the final definition of `Date` as a union of all possible notations.

Note that we employ a shorthand notation for the setup of constraints. `MonthDayYear` for example has to match the language defined by the union of the `MonthDay` domain and the concatenation of `MonthDay`, a separator, and the `Year`. This notation enables a more readable definition of constraint systems.

Table 3 shows a brief performance evaluation as done in Section 5.1. As can be seen, CLPQS is notably faster than ConString. The automata created by ConString are probably large due to the alternative choice operator and the union operator leading to a lack of performance when labeling data. Reducing the size of automata, e.g., by removing ϵ-transitions, will likely increase performance.

We also encoded the example in SMT-LIB to compare ConString and CLPQS with Z3-str3 and CVC4. Z3 found a single solution in around 0.070 seconds while CVC4 took around 0.084 seconds. Again, we were unable to compute multiple solutions with both solvers as they timed out.

5.3 Generation of Data in JSON

As a further and more involved example, we want to generate data describing different colors in JavaScript Object Notation (JSON). A color should be de-

Listing 7. An exemplary dataset in JSON containing the colors black and white.

```
1 { "colors": [
2   { "color": "black",
3     "code": { "rgb": [0,0,0], "hex": "#000000" } },
4   { "color": "white",
5     "code": { "rgb": [255,255,255], "hex": "#FFFFFF" } } ] }
```

scribed by a name, a six byte hexadecimal code and a corresponding RGB color code. An exemplary dataset in JSON containing the colors black and white is shown in Listing 7.

For the given example, we want to ensure that the hexadecimal and RGB code of a color match each other. Further, each color in the set of colors should be unique. The latter requirement entails the need of two further constraints not mentioned in this paper yet: First, we need to be able to state the difference between two string variables (`str_diff/2`). This is achieved by a propagation rule which is triggered if both string variables have been labeled, i.e., they hold constant values, and checks for exact inequality (`\==/2`) between both values. If both values are equal, ConString backtracks and searches for different values effectively restarting the computation from the last choicepoint. Second, we need to be able to state the difference of strings in between a list of string variables (`str_all_diff/2`). This is achieved by a simplification rule propagating pairwise inequality constraints for each pair of elements.

The constraint system used to generate datasets in JSON as described above is shown in Listing 8. First, we generate a given amount of hexadecimal color codes which have to be all different (lines 20 and 21). After labeling all hexadecimal color codes, we generate the corresponding RGB color codes using SWI-Prolog's predicate `hex_bytes/2`. We then use the labeled hexadecimal and RGB color codes to generate the strings describing a dataset entry, which is achieved by the predicate `get_color_entry/3`, and join all strings by concatenation (`list_of_colors_concat/3`). Finally, we further concatenate strings to the generated string concatenation describing single dataset entries (line 24) to obtain the desired data format in JSON. This last step shows the difference between test data generation and data structure generation. In theory, only the first is needed to gain sensible test data, as the results can easily be stored in various data structures depending on the requirements for the SUT. However, integrating data structure generation into the constraint problem would render data generation more self-contained and could thus be desirable for users. Both data generation and data structure generation are strictly split inside the constraint system, i.e., there are two distinct blocks of constraints which are labeled individually (see labelings in line 21 and 25 of Listing 8).

To evaluate the performance of ConString as done for the other case studies, we generate one dataset for varying amounts of dataset entries, i.e., colors. However, we noticed several performance bottlenecks in both, ConString and

Listing 8. The constraint system to generate datasets in JSON containing colors.

```
1  get_color_entry(Hex, RgbList, Color) :-
2      term_string(RgbList, Rgb1),
3      escape_special_characters(Rgb1, Rgb),
4      Prefix = "\\{\"color\":\"test\",\"code\":\\{\"rgb\":",
5      Color match Prefix + Rgb + ",\"hex\":#" + Hex + "\\}\\}".
6  list_of_colors_concat_acc([], [], Acc, Acc).
7  list_of_colors_concat_acc([Hex|HT], [Rgb|RT], Acc, Concat) :-
8      get_color_entry(Hex, Rgb, Color),
9      NewAcc = '+'(Acc, '+'(",", Color)),
10     list_of_colors_concat_acc(HT, RT, NewAcc, Concat).
11 list_of_colors_concat([], [], "").
12 list_of_colors_concat([Hex|HT], [Rgb|RT], Concat) :-
13     get_color_entry(Hex, Rgb, Color),
14     list_of_colors_concat_acc(HT, RT, Color, Concat).
15 list_of_hex_codes(0, []) :- !.
16 list_of_hex_codes(C, [HexCode|T]) :-
17     str_in(HexCode, "([A-F]|[0-9]){6}"),
18     C1 is C-1, list_of_hex_codes(C1, T).
19 json_colors(Amount, JSON) :-
20     list_of_hex_codes(Amount, LHex),
21     str_all_diff(LHex), str_label(LHex),
22     maplist(hex_bytes, LHex, RgbList),
23     list_of_colors_concat(LHex, RgbList, ColorsConcat),
24     JSON match "\\{\"colors\":\\[" + ColorsConcat + "\\]\\}",
25     str_label([JSON]).
```

Table 4. Benchmarks for generating datasets in JSON. Walltime in seconds.

Amount of Colors	1	2	3	4	5	10	50	100
ConString	0.005	0.003	0.004	0.006	0.007	0.021	3.878	93.196

CLPQS, when trying to benchmark the JSON generation. ConString displayed a quadratic increase in runtime with respect to the number of variables as can be seen in Table 4. autogen's runtime was somewhat erratic, i.e., it was sometimes faster for a higher (even) number of colors. Overall, autogen's runtime was less predictable than the one of ConString. We will discuss performance bottlenecks of both solvers and how they can be coped with in the following three paragraphs.

Order of Constraints Although constraint systems are declarative, the order of constraints influences performance. For instance, in the given example, we have to ensure that the hexadecimal and RGB color codes are constant values (lines 21 and 22) before setting up the concatenation constraints. Otherwise,

Listing 9. An example showing a possible bottleneck for performance when implementing `str_all_diff/2`.

```
1 str_in(X, "1|2"), str_in(Y, "1|2"),
2 str_in(Z, "[0-9]{0,1000}"),
3 str_all_diff([X,Y,Z]), str_label([X,Y,Z]).
```

performance drops drastically since large automata domains have to be created holding variable references. As soon as such a variable reference is labeled, concatenation constraints are triggered and automata have to be intersected with their prior automata domains (see Section 4.3, Listing 1) containing the unlabeled variable references, ultimately leading to bad performance. Note that within our framework, the intersection operation on finite automata is usually the most complex operation when solving string constraints. If we evaluate the concatenations after all necessary variables have been labeled (lines 23 and 24), no intersections have to be computed on automata domains.

Performance of String Difference Currently, `str_diff/2` is only triggered if both arguments are constant strings and does not propagate any knowledge to an unlabeled domain of a string variable. This is a bottleneck for performance when using the `str_all_diff/1` constraint. Since we only support a linear enumeration order by now, the solver has to backtrack a lot for large lists of variables between labeling a string and checking for inequality. If all variables have the same domain (e.g., as shown in Listing 8), the domain gets enumerated linearly for each variable in the list until finding a new value that is different to the ones labeled so far. For ConString, this leads to a runtime that grows quadratic with the size of the list of variables. As future work, we want to investigate propagating knowledge to domains instead of only checking inequality between constant strings. In SWI-Prolog's CLP(FD) library this corresponds to the constraints `all_different/1`, which behaves similar to our implementation, and `all_distinct/1`, which propagates knowledge to unlabeled domains.

In order to improve upon simple pairwise difference computation, it is essential to propagate `str_diff/2` as soon as the two involved variables are constant values instead of waiting for all variables to be labeled. For instance, consider Listing 9 with `X` and `Y` sharing a domain and `Z` whose domain is considerably larger. When labeling the variables using a linear enumeration order, the first assignment of `X` and `Y` is the same, i.e., the string "1". If the pairwise difference constraints are triggered after labeling, the equality of the first two variables is only identified after labeling all three variables, with the last choicepoint being in the labeling of `Z` although the variable is not involved in the conflict at all. The solver would enumerate the domain of `Z` exhaustively, before detecting the conflict. To counter this behaviour,one has to ensure that `str_diff/2` is triggered as soon as `X` and `Y` are labeled, possibly suspending an ongoing labeling of variables.

Data Generation vs. Data Structure Generation Generating a full data structure in JSON representation drastically increased the strain put on the constraint solvers. Within a single labeling operation, the combined generation of data and JSON representation caused a lot of unneeded backtracking through the two problems. With the two labeling operations split up, performance was increased while decreasing the declarativeness of the problem statement.

Overall, including the data structure generation in the constraint satisfaction problem lead to a severe performance decline. In consequence, we suggest splitting the generation of test data from storing it inside an appropriate data structure for testing. While this reduces self-containment of the encoding, it has several benefits as well:

- Constraints are considerably simpler, in particular, many concatenation constraints are avoided at all.
- Variables are less intertwined which reduces the evaluation time of consistency and propagation algorithms.
- Flexibility in the enumeration order is increased which could open the way for optimization.

5.4 Comparison to Test Data Generators

In the following we will give a brief comparison between data generation tools based on constraint solving, such as autogen and ConString, and typical test data generators, i.e., imperative implementations of enumeration algorithms. In this section we have seen three case studies in which we applied our approach of constraint logic programming to test data generation. While the IBAN example in Section 5.1 is motivated by a real world application (cf. Friske & Ehmke 2019 [10]), it can easily be replicated by a typical test data generator not using a declarative approach, as shown examplary in Listing 10. Such an IBAN generator would probably also keep a linear runtime depending on the number of generated IBANs, whereas we observed in Table 2 that ConString exhibits a non-linear growth. However, if one is in need of generating IBANs with a certain checksum for testing purposes, the test data generator in Listing 10 would need to be modified to account for the requirement. In contrast, with constraint programming, e.g., as used by ConString and autogen, additional requirements can be realized by simply adding a constraint, e.g., `CheckSum #= DesiredChecksum`.

The date example serves as data source to a common problem in programming, that of parsing date inputs (e.g., by the user via a text field). Although the generation itself can be done easily with a test data generator which randomly chooses a style, a weekday, and a calendar day, the implementation in Listing 6 can easily be improved to generate only valid dates (e.g., a correct weekday or matching calendar day per month) by adding some further specifications into the constraint system.

In our third example, the color database in JSON, we generate a more strictly defined data set. While the hexadecimal and the RGB color code in a single dataset entry must match, all colors in a generated dataset need to be exclusive.

Listing 10. Pseudocode of an IBAN test data generator.

```
bban = 100000000000000000
while bban <= 999999999999999999:
    bban_country = bban * 1000000 + 131400
    checksum = 98 - (bban_country mod 97 )
    iban = concat("DE", checksum, bban)
    bban += 1
    yield iban
```

In contrast to a classical imperative test data generator, in which one needs to keep track of generated colors explicitly, our constraint-based approach enables a more declarative implementation using difference constraints and backtracking.

In conclusion, traditional test data generators might run faster and can, depending on the use case, be more suitable than a constraint solver. For highly intertwined test data or requirements that are likely to change, a more declarative approach based on constraint solving leads to a clearer specification of the test data to be generated and allows for simple adaptation to requirement changes. As seen in the calendar date example, using a declarative approach allows constructing complex structures from simple building blocks. No further control structures or instructions are required besides describing the data format.

Due to the intended use for test data generation, we have the strong belief that such data driven development resonates more with the problem domain than using, e.g., imperative programming languages. Thanks to Prolog's off-the-shelf backtracking capabilities, exhaustively traversing a search space is provided by default and one does not depend on explicit loop constructs or caching of results: each solution is found exactly once. Another benefit is the separation of the definition of data and the search for solutions. Consider again the pseudocode example shown in Listing 10. The enumeration order of calculated IBANs will always be the same. To reach another order, the code again needs to be adapted. On the other hand, the implementation for ConString shown in Listing 5 is independent of any enumeration order. The order can easily be changed by passing a corresponding argument to `str_labeling/2` as outlined in Section 4.3.

6 Way Forward & Future Work

6.1 An Efficient Backend

For classic domain propagation to work on strings, an efficient representation of possible values is needed. So far, we represent automata as outlined in Section 4.2. As discussed, this is not the most efficient approach, as certain algorithms need to traverse the list of states or transitions to find a particular one.

Other known automaton libraries such as dk.brics.automaton [26] feature more efficient representations and algorithms. However, these are usually based on using pointers or objects and cannot easily be ported to Prolog for obvious

reasons. At the same time, connecting the Java or C ports of the library to our Prolog system leads to all kinds of difficulties when it comes to proper handling of backtracking. Moreover, Prolog programs are no longer declarative when using stateful data structures without cloning data after each operation.

As future work, we want to experiment with porting dk.brics.automaton or a comparable library to Prolog while retaining its efficiency. So far, we have different approaches in mind. First, we could implement low-level data structures outside of Prolog (e.g., in C) and render them backtrackable using a thin Prolog layer. Second, we could mimic the internal workings of the library, e.g., using attributed variables to store (mutable) class variables and links to other "objects". While this would avoid possible backtracking issues, it would not be as idiomatic. Furthermore, we want to evaluate whether it is more efficient to use deterministic finite automata or, in general, ϵ-free automata. Additionally, we do not provide options for labeling, e.g., concerning the enumeration order. Additional options like enumerating a string domain in alphabetical, reversed alphabetical or a randomized order will most likely improve performance for some constraint satisfaction problems. This would also enable to provide different distributions of test data for a given domain. Especially a randomized enumeration order enables the generation of more diverse test data. Yet, labeling options of integrated solvers like CLP(FD) can already be used.

6.2 Combining Solvers

Of course, a solver like the one we outlined above would still be too weak to efficiently support the constraints we discussed in Section 1. In consequence, we envision an integration of a CLP-based solver and the other solvers discussed in Section 3 into a combined solving procedure. This could be done following the approach we used for first-order logic in prior work [23].

A more simple strategy would be to use multiple solvers at once and returning the first result computed. This will have a performance benefit, given that the solvers described in Section 3 have diverse approaches and mixed performances in certain situations. Implementing such a portfolio is somewhat complicated, since there is no standardized interface for constraint solvers [25], leading to a large overhead translating constraints in between solvers. However, a promising draft for an interface [32] has been proposed recently.

7 Conclusion

In this paper, we discussed how synthetic test data can be generated and what the common pitfalls are. We discussed currently available solvers over strings and outlined that string constraint solving has made considerable progress recently. However, hurdles remain and generation of artificial test data remains complicated at least.

We implemented a simple prototype of a string constraint solver based on constraint logic programming and classical domain propagation. While it does

not yet offer all features desired, our prototype shows that our approach is feasible and promising.

However, we believe that no single solver will be able to handle all requirements sufficiently and that reimplementing features commonly found in other solvers might not be worthwhile. In consequence, we think that an integration of solvers such as the one discussed in Section 6.2 is very promising, and we hope to be able to lift our results for first-order-logic to string domains in the future.

References

1. P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Trau: SMT solver for string constraints. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–5, 2018.
2. P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Flatten and Conquer: A Framework for Efficient Analysis of String Constraints. In *Proceedings PLDI 2017*, pages 602–617. ACM, 2017.
3. R. Amadini, P. Flener, J. Pearson, J. D. Scott, P. J. Stuckey, and G. Tack. MiniZinc with Strings. *CoRR*, abs/1608.03650, 2016.
4. R. Amadini, G. Gange, P. Stuckey, and G. Tack. A Novel Approach to String Constraint Solving. pages 3–20, 08 2017.
5. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, July 2011.
6. M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *FMCAD*, pages 55–59. IEEE, 2017.
7. T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu. What Is Decidable about String Constraints with the ReplaceAll Function. *CoRR*, abs/1711.03363, 2017.
8. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
9. N. ElGamal, A. ElBastawissy, and G. Galal-Edeen. Data Warehouse Testing. In *Proceedings EDBT/ICDT*, EDBT '13, pages 1–8. ACM, 2013.
10. M. Friske and D. Ehmke. Modellbasierte Testdatenspezifikation und -generierung mittels Äquivalenzklassen und SQL. In *Proceedings TAV*, 02 2019.
11. T. Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(1–3):95–138, 1998.
12. X. Fu and C.-C. Li. A String Constraint Solver for Detecting Web Application Vulnerability. pages 535–542, 01 2010.
13. V. Ganesh and D. L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings CAV*, CAV'07, pages 519–531. Springer, 2007.
14. K. Golden and W. Pang. Constraint Reasoning over Strings. In F. Rossi, editor, *Proceedings CP*, pages 377–391. Springer, 2003.
15. F. Haftmann, D. Kossmann, and E. Lo. A framework for efficient regression tests on database applications. *The VLDB Journal*, 16(1):145–164, Jan 2007.
16. K. Houkjær, K. Torp, and R. Wind. Simple and Realistic Data Generation. In *VLDB*, 2006.

17. Information technology – Security techniques – Information security management systems – Requirements. Standard, International Organization for Standardization, Geneva, CH, 6 2017.
18. D. R. Jeske, P. J. Lin, C. Rendon, R. Xiao, and B. Samadi. Synthetic Data Generation Capabilties for Testing Data Mining Tools. In *Proceedings MILCOM*, pages 1–6, Oct 2006.
19. L. Karttunen, J.-P. Chanod, G. Grefenstette, and A. Schille. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328, 1996.
20. M. S. A. Khan and A. ElMadi. Data Warehouse Testing an Exploratory Study. Master's thesis, School of Computing, Blekinge Institute of Technology, Karlskrona, Sweden, 2011.
21. A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proceedings ISSTA 2009*, July 21–23, 2009.
22. E. K. Klaus Franz, Tanja Tremmel. *Basiswissen Testdatenmanagement: Aus- und Weiterbildung zum Test Data Specialist – Certified Tester Foundation Level nach GTB.* dpunkt, 2018.
23. S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B models. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.
24. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. pages 646–662, 07 2014.
25. T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters. An Efficient SMT Solver for String Constraints. *Form. Methods Syst. Des.*, 48(3):206–234, June 2016.
26. A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2017. `http://www.brics.dk/automaton/`.
27. A. Pretschner. Zum modellbasierten funktionalen Test reaktiver Systeme. 2003. `http://mediatum.ub.tum.de/doc/601738/000006bb.pdf`.
28. C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 2015, 2010.
29. J. Singh and K. Singh. Statistically Analyzing the Impact of Automated ETL Testing on the Data Quality of a Data Warehouse. *IJCEE*, 1(4):488–495, 2009.
30. A. Spillner and T. Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard.* dpunkt, 3. edition, 2005.
31. N. Tillmann and J. De Halleux. Pex: White Box Test Generation for .NET. In *Proceedings TAP*, TAP'08, pages 134–153. Springer, 2008.
32. C. Tinelli, C. Barret, and P. Fontaine. Unicode Strings (Draft 2.0), 2019. `http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml`.
33. M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings CCS*, CCS '14, pages 1232–1243. ACM, 2014.
34. M. Triska. The finite domain constraint solver of SWI-Prolog. In *Proceedings FLOPS*, volume 7294 of *LNCS*, pages 307–316, 2012.
35. M. Triska. The Boolean Constraint Solver of SWI-Prolog: System Description. In *Proceedings FLOPS*, volume 9613 of *LNCS*, pages 45–61, 2016.
36. M. Triska. Boolean constraints in SWI-Prolog: A comprehensive system description. *Science of Computer Programming*, 164:98–115, 2018.
37. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *CoRR*, abs/1011.5332, 2010.

38. Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3):249–288, 2017.
39. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings ESEC/FSE*, ESEC/FSE 2013, pages 114–124. ACM, 2013.