Answer-Set Programming as a new Approach to Event-Sequence Testing

Esra Erdem[1], Katsumi Inoue[2], Johannes Oetsch[3], Jörg Pührer[3], Hans Tompits[3], Cemal Yilmaz[1]

[1]*Sabanci University, Faculty of Engineering and Natural Sciences,*
Orhanli, Tuzla, Istanbul 34956, Turkey
Email: {esraerdem,cyilmaz}@sabanciuniv.edu

[2]*National Institute of Informatics,*
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
Email: ki@nii.ac.jp

[3]*Technische Universität Wien, Institut für Informationssysteme 184/3,*
Favoritenstraße 9-11, A-1040 Vienna, Austria
Email: {oetsch,puehrer,tompits}@kr.tuwien.ac.at

***Abstract*—In many applications, faults are triggered by events that occur in a particular order. Based on the assumption that most bugs are caused by the interaction of a low number of events, Kuhn et al. recently introduced *sequence covering arrays* (SCAs) as suitable designs for event sequence testing. In practice, directly applying SCAs for testing is often impaired by additional constraints, and SCAs have to be adapted to fit application-specific needs. Modifying precomputed SCAs to account for problem variations can be problematic, if not impossible, and developing dedicated algorithms is costly. In this paper, we propose *answer-set programming* (ASP), a well-known knowledge-representation formalism from the area of artificial intelligence based on logic programming, as a declarative paradigm for computing SCAs. Our approach allows to concisely state complex coverage criteria in an *elaboration tolerant* way, i.e., small variations of a problem specification require only small modifications of the ASP representation.**

***Keywords*-event-sequence testing; combinatorial interaction testing; answer-set programming.**

I. Introduction

In many applications, faults only show up if events occur in a certain order. An example are atomicity violations in multi-threaded applications where a pair of shared memory accesses of one thread is interleaved with an unfortunate access of another thread. Testing such applications thus requires exercising *event sequences*. Since the number of event sequences is factorial in the number of events, exhaustive testing is infeasible in general. If we assume that bugs are triggered by the interaction of only a low number of events—this is empirically supported by respective bug reports—, testing costs can be reduced drastically without sacrificing much fault-detection potential by using suitable combinatorial designs [1], [2]. To this end, Kuhn et al. [3], [4] introduced *sequence covering arrays* (SCAs) for combinatorial event sequence testing. An SCA is an array of permutations of events such that any t events, possibly interleaved with other events, will be tested in every t-way order at least once. SCAs are relevant in scenarios where the order of events is decisive, like testing of user-interfaces, dynamic web applications, method calls for unit-testing, or multi-threaded programs.

In practice, a direct application of SCAs for testing is often impaired by additional constraints on the order of events. Also, the conditions that identify the sequences that should be covered can vary and often involve quite complex definitions. For example, to test thread interleavings, one could require to test all sequences such that one variable is written by one thread and subsequently read by another thread such that there is no write operation between them [5], [6].

One approach to address such considerations is to accordingly modify precomputed SCAs as exemplified by Kuhn et al. [3], [4]. This means that any test sequence which, e.g., violates some ordering constraints has to be removed from the SCA. To maintain coverage, removed sequences have to be replaced by permutations thereof that comply to the problem specific requirements. This is not always possible in a straightforward way and can result in a considerable and in principle avoidable overhead regarding the size of arrays. On the other hand, developing and maintaining dedicated algorithms to compute variations of SCAs usually comes with high costs and is not preferable if requirements change over time or one wants to experiment with different designs.

We propose to use *answer-set programming* (ASP) [7] for computing SCAs and variations thereof. ASP is a genuine declarative programming paradigm where a problem is encoded by means of a logic program such that the solutions of a problem correspond to the models, called *answer sets*, of the program. On the one hand, as an expressive high-level specification language, it allows to state complex coverage criteria, involving constraints and complex, possibly recursive, definitions, in a concise and *elaboration-tolerant* way, i.e., small variations in a problem specification require only small modifications of the program representation. On the other hand, SCAs can be efficiently computed through highly

optimised ASP solvers [8]. Since it requires only little effort to state quite complex coverage conditions in ASP, a tester is able to rapidly specify different versions of SCAs.

This paper is organised as follows. In Section II, we review SCAs and ASP. Then, we show how SCAs can be generated using ASP in Section III. We present improved, sometimes optimal, upper bounds regarding the size of many SCAs. We furthermore present a greedy algorithm, based on ASP, for computing larger SCAs. In Section IV, we turn towards a real-world example described by Kuhn et al. [3], [4]. We discuss how the basic ASP encoding from Section III can be refined to take different constraints and problem variations into account. Finally, we discuss related work in Section V and conclude in Section VI.

II. Preliminaries

In this section, we review the formal definition of SCAs and give a brief background on ASP.

A. Sequence Covering Arrays (SCAs)

SCAs, introduced by Kuhn et al. [3], [4], are combinatorial designs related to covering arrays. While covering arrays require that each t-way combination of parameters occurs at least once in a test case for some fixed t, SCAs take the order of events into account and require that each t-sequence of events is tested in at least one test sequence in that order, where a *t-sequence* over a set S of symbols is a t-tuple of pairwise distinct elements of S. Following Kuhn et al. [3], [4], we formally define SCAs as follows.

Definition 1: A *sequence covering array* (SCA) with parameters n, S, and t, or an (n, S, t)-SCA for short, is an $n \times |S|$ matrix M of symbols from a finite set S of symbols such that (i) each row of M is a permutation of S and (ii) for each t-sequence $\sigma = (s_1, s_2, \ldots, s_t)$ over S, there is at least one row $\varrho = (a_{i1}, \ldots, a_{i|S|})$ in M such that σ is a subsequence of ϱ.

We say that an (n, S, t)-SCA is of *strength* t and of *size* n. The *sequence covering array number* $\mathrm{SCAN}(S, t)$ is the smallest n such that an (n, S, t)-SCA exists. An (n, S, t)-SCA is *optimal* if $\mathrm{SCAN}(S, t) = n$. We will also denote an $(n, \{1, \ldots, s\}, t)$-SCA as an (n, s, t)-SCA with $\mathrm{SCAN}(s, t)$ for brevity.

For illustration, the following matrix M constitutes an optimal $(7, 5, 3)$-SCA:

$$M = \begin{pmatrix} 5 & 2 & 3 & 1 & 4 \\ 3 & 2 & 5 & 4 & 1 \\ 1 & 5 & 4 & 3 & 2 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 2 & 5 & 1 & 3 \\ 2 & 4 & 3 & 1 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

Each of the 7 rows is a permutation of the set $S = \{1, \ldots, 5\}$ and each 3-sequence over S is covered by at least one row. For instance, the 3-sequence $(5, 3, 4)$ is covered by the first row of M. Note that there are $5 \cdot 4 \cdot 3 = 40$ such 3-sequences.

A collection of precomputed SCAs of strength 3 and 4, involving 5 to 80 events, is available online [9].These SCAs were computed using a simple greedy algorithm introduced by Kuhn et al. [3], [4]. Note that this algorithm is the only approach for computing SCAs implemented so far. To compute a t-strength SCA for a set S of events, this algorithm iteratively computes single rows of the SCA: It computes a fixed number of permutations of S. Then, it selects the permutation π that obtains maximal coverage of previously uncovered t-sequences as the next row of the SCA. After that, π in reverse order, π', is added. Adding π' is justified because π' always covers the same number of previously uncovered t-sequences as π [4]. This procedure is iterated until all t-sequences are covered.

One downside of this greedy algorithm is that additional constraints on the order of events arising from the requirements of different test scenarios are hard to incorporate. To overcome this shortcoming, we use ASP in what follows as a declarative tool to compute SCAs and demonstrate that quite complex constraints can be incorporated into a solution in a concise and elaboration-tolerant way, and with ease.

B. Answer-Set Programming (ASP)

ASP [7] is a relatively new declarative programming paradigm. The underlying idea of ASP is to declaratively represent a computational problem as a logic program whose models, called "answer sets", correspond to the solutions, and to find the answer sets for that program using an ASP solver. Due to the expressiveness of ASP that allows to represent, for instance, aggregates and recursive definitions, and due to the continuous improvements of the efficiency of ASP solvers, such as `clasp` [10], we argue that ASP can efficiently and effectively be used to compute SCAs. Indeed, ASP has been used in a wide range of applications from different fields, such as semantic-web reasoning, systems biology, planning, diagnosis, information integration, configuration, multi-agent systems, cladistics, and super optimisation. For a comprehensive introduction to ASP, we refer to the textbook by Baral [7].

We recapitulate the basic elements of ASP in the following. An *answer-set program* is a finite set of rules of the form

$$a_0 :- a_1, \ldots, a_m, \mathrm{not}\ a_{m+1}, \ldots, \mathrm{not}\ a_n, \qquad (1)$$

where $n \geq m \geq 0$, a_0 is a propositional atom or $\bot$, and all $a_1, \ldots, a_n$ are propositional atoms; the symbol "not" denotes *default negation*. If $a_0 = \bot$, then Rule (1) is a *constraint* (in which case a_0 is usually omitted). The intuitive reading of a rule of form (1) is that whenever $a_1, \ldots, a_m$ are known to be true and there is no evidence for any of the default negated atoms $a_{m+1}, \ldots, a_n$ to be true, then a_0 has to be true as well. Note that $\bot$ can never become true.

An *answer set* for a program is defined following Gelfond and Lifschitz [11]. An *interpretation* I is a finite set of propositional atoms. An atom a is *true* under I if $a \in I$, and *false* otherwise. A rule r of form (1) is true under I if $\{a_1, \ldots, a_m\} \subseteq I$ and $\{a_{m+1}, \ldots, a_n\} \cap I = \emptyset$ implies $a_0 \in I$. Interpretation I is a *model* of a program P if each rule $r \in P$ is true under I. Finally, I is an answer set of P if I is a subset-minimal model of P^I, where P^I is defined as the program that results from P by deleting all rules that contain a default negated atom from I, and deleting all default negated atoms from the remaining rules.

Programs can yield no answer set, one answer set, or many answer sets. For instance, the program

$$\{p :- \text{ not } q,\ q :- \text{ not } p\} \tag{2}$$

has two answer sets: $\{p\}$ and $\{q\}$.

When we represent a problem in ASP, some rules "generate" answer sets corresponding to "possible solutions", and some "eliminate" the answer sets that do not correspond to solutions. The rules in program (2) are of the former kind; constraints are of the latter kind. For instance, adding the constraint $\bot :- p$ to a program P eliminates all answer sets of P containing p. In particular, adding $\bot :- p$ to program (2) eliminates the answer set $\{p\}$.

When we represent a problem in ASP, we often use special constructs of the form $l\{a_1, \ldots, a_k\}u$ (called *cardinality expressions*) where each a_i is an atom and l and u are nonnegative integers denoting the *lower bound* and the *upper bound* of the cardinality expression [12]. Such an expression describes the subsets of the set $\{a_1, \ldots, a_k\}$ whose cardinalities are at least l and at most u. In heads of rules, cardinality expressions generate answer sets containing subsets of $\{a_1, \ldots, a_k\}$ whose cardinality is at least l and at most u. When used in constraints, they eliminate answer sets that contain such respective subsets.

A group of rules that follow a particular pattern can often be described in a compact way using *schematic variables*. For instance, we can write the program $p_i :- \text{ not } p_{i+1}$, $(1 \leq i \leq 7)$ as follows:

$$index(1),\ index(2), \ldots, index(7),$$
$$p(i) :- \text{ not } p(i+1), index(i).$$

ASP solvers compute an answer set for a given program that contains variables after "grounding" the program, e.g., by the grounder `gringo` [13]. A grounder systematically replaces each rule r with variables by its ground instances that result from r by uniformly replacing each variable by constants from the program. Variables can also be used "locally" to describe a list of literals. For instance, the rule $1\{p_1, \ldots, p_7\}1$ can be represented as $1\{p(i) : index(i)\}1$.

In addition to the constructs above, current state-of-the-art ASP solvers support many language extensions like *functions*, *built-in arithmetics*, *comparison predicates*, *aggregate atoms*, *maximisation* and *minimisation statements*, as well as *weak constraints*.

In the remainder of this paper, we use the syntax that is supported by the solver `clasp` along with the grounding tool `gringo` when presenting programs [14].

For illustrating problem solving in ASP, consider the following encoding of the 3-colorability problem (3COL):

```
colour(red;green;blue).
1{asgn(N,C):colour(C)}1 :- node(N).
:- edge(X,Y), asgn(X,C), asgn(Y,C).
```

The first rule abbreviates three facts that state that red, green, and blue are colours, respectively. The second rule is a choice rule. Its intuitive reading is that if `N` is a node, then both an upper bound and a lower bound on the number of colours assigned to this node, expressed by `asgn(N,C)`, is 1. This means that each node gets assigned precisely one colour from the set of available colours defined by `colour/1`. The last rule is a constraint that forbids that there is an edge between any two nodes with the same colour. If the above program is joined with facts over `edge/2` and `node/1` that represent a graph G, the answer sets correspond one-to-one to the valid 3-colourings of G.

Sometimes, one is not only interested in arbitrary solutions to a problem but in solutions that are optimal according to some preference relation. ASP solvers like `clasp` support optimisation statements that allow to express such preferences. For illustration, assume that, for some reason, we want to minimise the number of blue nodes in the above 3COL example. This can be expressed by simply adding the following minimise statement:

```
#minimize[asgn(N,blue):node(N)].
```

The meaning of such a statement is that `clasp` computes answer sets where the sum of literals `asgn(N,blue)`, where `N` is a node, is minimal among all answer sets.

III. SCA Computation

We now discuss how ASP can be used to generate SCAs. Our goal is not only to present approaches to compute generic SCAs, i.e., SCAs created without additional constraints or requirements, rather we want to demonstrate that ASP can be used as an efficient and effective declarative tool to compute SCAs tailored to specific test scenarios.

Ahead of our discussion in Section IV, addressing how different problem elaborations can be incorporated into a single answer-set program, we introduce an answer-set program for computing generic SCAs. We also introduce a new greedy approach that combines a simple variation of the basic ASP encoding with an iterative greedy procedure.

A. Basic Encoding

We first present an ASP program for computing (n, s, t)-SCAs with $t = 3$. We assume throughout that $s \geq 2$. Note that this program can be changed in a straightforward way

```
% guess sequence covering array
 sym(1..s). row(1..n).
 1{first(N,S):sym(S)}1 :- row(N).
1{next(N,S,T):sym(T)}1 :- first(N,S).
0{next(N,T,U):sym(U)}1 :- next(N,_,T).

% the happens-before relation
hb(N,X,Y) :- next(N,X,Y).
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).

% each symbol occurs once in each row
:- hb(N,S,S).
:- row(N), sym(S), first(N,T), S!=T,
   not hb(N,T,S).

% check if each 3-sequence is covered
threeSeq(X,Y,Z) :- sym(X;Y;Z),X!=Y,Y!=Z,X!=Z.
covered(X,Y,Z)  :- hb(N,X,Y), hb(N,Y,Z).
:- threeSeq(X,Y,Z), not covered(X,Y,Z).
```

Figure 1. ASP encoding $\Pi^3(n,s)$.

to obtain encodings for any fixed $t > 3$. An encoding for SCAs where t is not fixed can be obtained using *disjunctive ASP*—this is however beyond the scope of this paper.

1) Encoding: We start by expressing that the symbols of the array are integers between 1 and `s`, and row indices of the SCA correspond to integers 1 to `n`. Note that `s` and `n` function as parameters of the program:

```
sym(1..s). row(1..n).
```

For the representation of the SCA, we use the predicate `next(N,X,Y)` expressing that in row `N` symbol `Y` is the direct successor of `X`. We next state that in any row `N` (i) one symbol `S` occurs first, (ii) the first symbol `S` in row `N` has a direct successor `T`, and (iii) if `T` is consecutive to `S`, then there is at most one symbol `U` that is consecutive to `T`:

```
 1{first(N,S):sym(S)}1 :- row(N).
1{next(N,S,T):sym(T)}1 :- first(N,S).
0{next(N,T,U):sym(U)}1 :- next(N,_,T).
```

So far, the above conditions are only necessary conditions for an $(n,s,3)$-SCA. We need further rules to guarantee that any row is a permutation of the symbols $\{1,\ldots,s\}$ and that coverage of all 3-sequences is achieved. We proceed by formalising the *happens-before* relation between two events. In particular, that one event symbol `X` occurs before another symbol `Y` in row `N` is represented by predicate `hb(N,X,Y)`, which is simply the transitive closure of the `next/3` relation:

```
hb(N,X,Y) :- next(N,X,Y).
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).
```

Directly expressing inductive definitions as above is a particular strength of ASP. Based on the happens-before relation, we can quite easily state that each event symbol has to occur precisely once in each row. We express this by means of two constraints. The reading of the first one is that it is forbidden that there is a row `N` such that a symbol `S` occurs before itself. The second constraint ensures that it is forbidden that there is a row `N` such that some symbol `S` different from the first symbol `T` does not occur after `T`. Together, the constraints imply that `next/3` indeed represents permutations.

```
:- hb(N,S,S).
:- row(N), sym(S), first(N,T), S!=T,
   not hb(N,T,S).
```

It only remains to require that each 3-sequence of symbols is covered by some row. We use predicate `threeSeq(X,Y,Z)` to represent the 3-sequences that we want to cover. A 3-sequence is simply a 3-tuple of pairwise distinct symbols:

```
threeSeq(X,Y,Z) :- sym(X;Y;Z),X!=Y,Y!=Z,X!=Z.
```

A 3-sequence $(\mathtt{X},\mathtt{Y},\mathtt{Z})$ is covered if `X` happens before `Y` and `Y` happens before `Z` in some row `N`. We finally define covered 3-sequences and forbid that a 3-sequence is not covered:

```
covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
:- threeSeq(X,Y,Z), not covered(X,Y,Z).
```

The entire ASP program $\Pi^3(n,s)$ with parameters n and s for generating $(n,s,3)$-SCAs is given in Figure 1.

Intuitively, each answer set of program $\Pi^3(n,s)$ represents an $(n,s,3)$-SCA. In fact, the answer sets of $\Pi^3(n,s)$ and the $(n,s,3)$-SCAs are in a one-to-one correspondence. This relation can be formalised as follows:

Definition 2: An answer set X of $\Pi^3(n,s)$, for $s \geq 2$, *represents* an $n \times s$ matrix M iff for any i, $1 \leq i < s$, and any r, $1 \leq r \leq n$, $M_{r,i} = s_1$ and $M_{r,i+1} = s_2$ precisely in case X contains the atom $\mathtt{next}(r,s_1,s_2)$.

Proposition 1: Each answer set of $\Pi^3(n,s)$ represents a single $(n,s,3)$-SCA, and each $(n,s,3)$-SCA is represented by a single answer set of $\Pi^3(n,s)$.

For illustration, to compute a $(7,5,3)$-SCA, `gringo` and `clasp` can be invoked as follows:

```
gringo sca-3.gr -c n=7,s=5 | clasp.
```

File `sca-3.gr` contains program $\Pi^3(n,s)$. The `gringo` option `-c n=7,s=5` instantiates the program parameters n and s to 7 and 5, respectively. Any resulting answer set corresponds to a $(7,5,3)$-SCA. For instance, in some answer set, the first row of the SCA M given in Section II-A is encoded by the atoms `next(1,5,2),next(1,2,3),next(1,3,1),next(1,1,4)`. To compute more than one $(7,5,3)$-SCA, an upper bound on the number of answer sets that `clasp` should compute can be specified as an integer option (0 means that all answer sets are computed).

2) Discussion: Program $\Pi^3(n,s)$ nicely illustrates how challenging search problems can be concisely encoded using ASP: The program consists of only 12 rules that closely reflect the problem statement in natural language. We note that only little training time is needed to enable a tester to use ASP for test authoring. This is mainly because of the

Table I
UPPER BOUNDS FOR $\mathrm{SCAN}(s,3)$ OBTAINED BY KUHN ET AL. AND OUR ASP ENCODING. A STAR INDICATES AN OPTIMAL BOUND.

s	n (Kuhn et al.)	n (ASP)
5	8	7^*
6	10	8^*
7	12	8^*
8	12	8^*
9	14	9
10	14	9
11	14	10
12	16	10
13	16	10
14	16	10
15	18	10
16	18	11
17	20	11

genuine declarative nature of ASP, which does not require specialised knowledge on data structures or algorithms. A more experienced ASP user needs about 15 minutes to develop a program such as the one given in Figure 1.

Also, by using our ASP encoding $\Pi^3(n,s)$ and the ASP solver `clasp`, we could improve known upper bounds for many SCAs significantly. A comparison of the SCAs generated using ASP and the greedy algorithm of Kuhn et al. is given in Table I. The SCAs that we have computed using ASP are publicly available [15]. Computation times for the reported upper bounds range from fractions of a second to about 20 minutes. We have considered strength 3 SCAs for 5 to 17 events. The known upper bounds reported by Kuhn et al. [3], [4] could be improved throughout. The more events are considered, the more drastic are the improvements; e.g., for 17 events, we need 45% less test sequences.

For small SCAs—viz. for 5 to 8 events—the new upper bounds are actually optimal bounds. Optimality of upper bounds was established using ASP itself. To show that an (n,s,t)-SCA is optimal, we try to compute an $(n-1,s,t)$-SCA. If this fails, i.e., the ASP solver terminates without returning an answer set, the (n,s,t)-SCA is indeed optimal. Since $\mathrm{SCAN}(8,3)$=8, 8 is a trivial lower bound for any $\mathrm{SCAN}(s,3)$ with $s>8$. Note that greedy algorithms, or any approaches based on incomplete search, are unable to prove optimal bounds or to establish lower bounds at all.

A limitation of using the ASP encoding $\Pi^3(n,s)$ concerns scalability. Though memory usage is always limited by a polynomial with respect to the input parameters n and s, the runtime of `clasp` is worst-case exponential for encoding $\Pi^3(n,s)$. On the other hand, the greedy approach of Kuhn et al. seems to scale quite well; the authors report on SCAs of strength three and four for up to 80 events [4].

B. Greedy Algorithm

In the remainder of this section, we introduce and discuss an ASP-based greedy algorithm, inspired by that of Kuhn et al. [3], [4], for computing larger SCAs. The motivation to study such an algorithm is to combine the modelling capabilities of ASP, especially in the light of constraints and problem elaborations (as detailed in the next section), with the scalability of a greedy approach.

Require: s is the number of symbols.
Ensure: N represents an $(n,s,3)$-SCA.
1: $N \Leftarrow \emptyset$
2: $n \Leftarrow 0$
3: **repeat**
4: $\quad n \Leftarrow n+1$
5: $\quad X \Leftarrow$ answer set of $\Pi^3_{grdy}(s,n) \cup N$
6: $\quad N \Leftarrow N \cup X|_{\texttt{next/3}}$
7: **until** N represents an $(n,s,3)$-SCA

Figure 2. Greedy algorithm for computing an $(n,s,3)$-SCA.

In this context, we also mention that the greedy algorithm of Kuhn et al. has a certain weakness, which is related to the heuristic that for any newly computed sequence the reverse sequence is added as well (cf. Section II). As we will show next, this makes the algorithm inherently unable to compute optimal SCAs in general. Actually, the inability to find optimal SCAs follows immediately from the observation that some optimal SCAs, e.g., $(7,5,3)$-SCAs, are of odd size. However, ASP can be used to show that even optimal SCAs of even size cannot be found by that greedy approach in general. The idea is to augment program $\Pi^3(n,s)$ by a rule that states that every second row is the inversion of the previous one. This is simply expressed by the following rule:

```
next(N,S,T):- row(N),next(N-1,T,S),N#mod2==0.
```

Here, predicate `#mod` is the usual modulo operation. Hence, the intuitive reading of this rule is that for any row with even index `N`, the `next` relation is the inverse of the `next` relation of the preceding row `N-1`. We know already from Table I that any $(8,6,3)$-SCA is optimal. However, $\Pi^3(8,6)$ augmented by the above rule yields no answer set, which shows that $(8,6,3)$-SCAs cannot be computed by the greedy algorithm of Kuhn et al. [3], [4]. Next, we present an ASP-based greedy algorithm inspired by that of Kuhn et al. that does not rely on adding inverted rows.

1) Encoding: Figure 2 represents our ASP-based greedy algorithm for computing SCAs. The main idea is to compute one row of a SCA at a time instead of computing the entire array. In each iteration, one further row is computed using ASP where the number of covered 3-sequences is maximised. For this purpose, we use program $\Pi^3_{grdy}(s,n)$, which is depicted in Figure 3. Program $\Pi^3_{grdy}(s,n)$ takes the number s of events and a row index n as parameters. Both the ASP encoding and the greedy algorithm are introduced only for SCAs of strength 3. However, versions for computing SCAs of strength greater than 3 are obtained in a straightforward way. To obtain a program for strength 4 SCAs, for example, only the last two rules of $\Pi^3_{grdy}(s,n)$ have to be replaced by the following two rules:

```
covered(W,X,Y,Z) :- hb(n,W,X), hb(n,X,Y),
                    hb(n,Y,Z).
```

```
% guess single SCA row with index n
sym(1..s).
1 {first(n,S)  : sym(S)} 1.
1 {next(n,S,T) : sym(T)} 1 :- first(n,S).
0 {next(n,S,T) : sym(T)} 1 :- next(n,_,S).

% the happens-before relation
hb(N,X,Y) :- next(N,X,Y).
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).

% each symbol occurs once in each row
:- hb(S,S).
:- sym(S), first(n,T), S!=T, not hb(n,T,S).

% maximize coverage
covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
#maximize[covered(_,_,_)].
```

Figure 3. ASP encoding $\Pi^3_{grdy}(s,n)$.

```
#maximize[covered(_,_,_,_)].
```

Program $\Pi^3_{grdy}(s,n)$ is quite similar to $\Pi^3(n,s)$. However, each answer set of $\Pi^3_{grdy}(s,n)$ corresponds only to a single row with index n of an SCA. The idea is to represent preceding rows with index 1 to $n-1$ by means of facts `next/3`. These facts are joined with $\Pi^3_{grdy}(s,n)$. Then, the answer sets of $\Pi^3_{grdy}(s,n)$ correspond to those rows that obtain maximal coverage of previously uncovered 3-sequences. The encoding follows the *guess, check, and optimise pattern*, hence we use guessing rules to span the search space, constraints to filter unwanted solution candidates, and rules that express a preference relation on answer sets. In particular, rule

```
#maximize[covered(_,_,_)].
```

states that we seek for answer sets with a maximal number of covered 3-sequences.

The algorithm itself is rather simple, see Figure 2: It takes parameter s as input and computes an $(n,s,3)$-SCA. Initially, the set N that represents a (partial) SCA by means of facts `next/3` equals the empty set. In each iteration, $\Pi^3_{grdy}(s,n) \cup N$ are used to compute the next row of the SCA that obtains maximal increase of previously uncovered 3-sequences. The respective `next/2` facts for that row are then added to N. This procedure iterates until no uncovered 3-sequences are left (the ASP solver itself will indicate that no further optimisation is possible). Since the computation of optimal answer sets can become very time consuming, we additionally impose an upper bound on the time that is spent for optimising answer sets, thus improvements in each step will not be maximal in general. However, this seems to be a reasonable compromise regarding runtime and the size of computed SCAs. The time limit for computing a single row ranged from 10 seconds to several minutes, depending on the problem size.

Table II
COMPARISON OF OUR GREEDY ASP APPROACH AND THAT OF KUHN ET AL. [3], [4]: UPPER BOUNDS FOR $\mathrm{SCAN}(s,3)$ AND $\mathrm{SCAN}(s,4)$.

s	$t=3$		$t=4$	
	Kuhn et al.	ASP	Kuhn et al.	ASP
10	14	11	72	55
20	22	19	134	104
30	26	23	166	149
40	32	27	198	181
50	34	31	214	-
60	38	34	238	-
70	40	36	250	-
80	42	38	264	-

2) Discussion: Table II summarises a comparison of our greedy ASP algorithm with the greedy algorithm of Kuhn et al. [3], [4] for strength 3 and 4 SCAs involving 10 to 80 events. For strength 3 SCAs, our algorithm is competitive with that of Kuhn et al. and upper bounds could be improved throughout by some rows. For strength 4 SCAs, the greedy ASP approach is feasible for up to 40 symbols where upper bounds could be improved even more drastically than for strength 3 SCAs. However, we were not able to compute SCAs for 40 to 80 symbols, which shows a limitation of our ASP-based approach that is probably acceptable unless the need for larger instances with a high level of interaction is indeed motivated by some application scenario. This limitation basically comes from the huge number of 4-sequences that need to be covered and that are represented by the program. Here, it is to mention that scalability is certainly a characteristic strength of the simple greedy algorithm of Kuhn et al., since dedicated data structures, e.g., efficient bit-vectors, can be used for representing covered sequences. However, by using ASP we get better bounds for 3-SCAs for up to 80 symbols and can also improve bounds for 4-SCAs for up to 40 symbols. Again, we emphasise that our goal is not to compute generic SCAs but to allow a tester to express different requirements with little effort, by adding or changing some rules of the ASP program, which can readily be done using the greedy ASP approach. We pursue this issue in the next section.

IV. PROBLEM ELABORATIONS

Next, we turn to the actual strengths of using ASP as an elaboration tolerant representation formalism for event sequence testing. We describe how ASP can be used for generating SCAs in a scenario that involves additional constraints and other problem variations that make it impossible to directly use precomputed SCAs. In particular, we use a *real-world testing problem* described by Kuhn et al. [3], [4] for making our point. The specification of this testing problem is as follows: There are 5 different devices that have to be connected to a laptop. These devices can be connected before or after a boot-up phase. Further actions that have to be performed on the laptop are opening an application and initiating a scanning process. The peripherals can be

connected to the laptop in any order; however, the order of events influences the functionality of the system. Thus, SCAs lend themselves as a basis for a suitable testing plan.

There are 8 events relevant for testing: connecting devices (`p1`, ..., `p5`), booting the system (`boot`), starting an application (`appl`), and running a scan (`scan`). Testing in this scenario is rather time consuming since it requires setting up the system manually. Therefore, obtaining an optimal test plan is a clear desideratum. Following Kuhn et al., only SCAs of strength 3 are considered to keep the size of the test plan reasonable.

A. Forbidden Sequences

For 8 events, optimal SCAs of strength 3 comprise 8 rows. However, we cannot use precomputed $(8, 8, 3)$-SCAs since certain constraints regarding the order of events have to be taken into account. While most events can happen in any order, starting the application cannot happen before the system is booted, and running a scan requires that the application is already running.

1) Encoding: Instead of covering all 3-sequences, we want to generate SCAs such that (i) in each row, `boot` happens before `appl` and `appl` happens before `scan`, and (ii) all 3-sequences such that `boot` happens before `appl` and `appl` happens before `scan` are covered by at least one row. We only have to slightly modify program $\Pi^3(n, s)$ to account for (i) and (ii). First, instead of integers to denote events, we would like to use more descriptive constant symbols. Thus, we replace `sym(1..s)` in $\Pi^3(n, s)$ by

```
sym(boot; p1; p2; p3; p4; p5; appl; scan).
```

Concerning (i), we define which orderings are excluded and add a respective constraint that forbids that event a happens before b if "a before b" is excluded.

```
excluded(scan,appl).
excluded(appl,boot).
excluded(X,Z) :- excluded(X,Y),excluded(Y,Z).
:- hb(_,X,Y), excluded(X,Y).
```

Regarding (ii), we simply define those 3-sequences that are not consistent with the excluded orderings as already covered:

```
covered(X,Y,Z) :- excluded(X,Y), sym(X;Y;Z).
covered(X,Y,Z) :- excluded(X,Z), sym(X;Y;Z).
covered(X,Y,Z) :- excluded(Y,Z), sym(X;Y;Z).
```

We denote the resulting program as $\Pi_1^3(n)$.

2) Discussion: Recall that for 8 symbols, $(8, 8, 3)$-SCAs are optimal. Since, $\Pi_1^3(8)$ does not yield any answer set, it follows that the stipulation on admissible orderings requires additional rows. In this case, this is because the number of 3-sequences that can be covered by a single row is reduced if certain events are required to happen in a strict order. Indeed, a solution for $\Pi_1^3(9)$ can be computed, hence 9 is an optimal bound for an SCA satisfying that each row is consistent with the specified ordering constraints. The solver `clasp` needs fractions of a second to find an SCA of size 9 and about one minute for checking optimality.

B. Redundant Sequences

Besides forbidden orderings, we also have to deal with redundant sequences: If devices are connected to the laptop before the boot-up phase, the order is not relevant. In fact, we only require strength 3 coverage for events `p1`, ..., `p5`, `appl`, and `scan`. Concerning the interaction of events `p1`, ..., `p5`, and `boot`, we regard strength 2 coverage as sufficient, i.e., we are only interested in whether the connection of the peripherals happens before or after the boot-up phase. Hence, we need a variable strength SCA, in which we seek to have strength 2 coverage for one set of events and strength 3 coverage for another one.

1) Encoding: First, we add two sets of facts to declare the sets of events for which we want to obtain strength 2 and strength 3 coverage, respectively:

```
threeWay(p1; p2; p3; p4; p5; appl; scan).
twoWay(boot; p1; p2; p3; p4; p5).
```

Next, we have to modify some rules where appropriate. In particular, we only want to cover 3-sequences over symbols from `threeWay/1`. Hence, we rewrite rule

```
threeSeq(X,Y,Z) :- sym(X;Y;Z),X!=Y,Y!=Z,X!=Z.
```

into

```
threeSeq(X,Y,Z) :- threeWay(X;Y;Z),
                   X!=Y, Y!=Z, X!=Z.
```

To address 2-way coverage of the symbols from `twoWay/1`, we add two further rules:

```
covered(X,Y) :- hb(_,X,Y).
:- twoWay(X;Y), X != Y, not covered(X,Y).
```

The resulting program is denoted by $\Pi_2^3(n)$.

2) Discussion: Program $\Pi_2^3(n)$ incorporates both forbidden configurations and redundant sequences. Respective SCAs can be obtained for $n = 8$ already. SCAs of size 8 are indeed optimal arrays, which follows from the observation that $\Pi_2^3(7)$ yields no answer set at all. It takes on average 0.1 seconds to compute the first answer set of a size 8 SCA when using `clasp` as ASP solver. Showing optimality, i.e., that no size 7 SCA exists, needs several minutes.

The solution approach of Kuhn et al. uses a pre-computed $(12, 7, 3)$-SCA to account for the seven events `p1`, ..., `p5`, `scan`, and `appl`. In a post-processing step, rows that are not consistent with the ordering constraints (cf. Section IV-A) are replaced. However, this requires that further rows are added to preserve coverage. Then, in a further manual post-processing step, to account for the two-way coverage with respect to events `p1`, ..., `p5`, and `boot`, Kuhn et al. add `boot` as the first event of each row. Finally, an additional row is added, in which all events `p1`, ..., `p5` are arranged prior to `boot`, thereby obtaining strength 2 coverage between

Table III
TEST PLAN OF SIZE 8 FOR THE LAPTOP APPLICATION OBTAINED FROM AN ANSWER SET OF $\Pi_4^3(8)$.

row	event 1	event 2	event 3	event 4	event 5	event 6	event 7	event 8
1	p3(l)	p2(r)	p1(b)	p4	boot	appl	scan	p5
2	boot	p4	p1(r)	appl	p5	p3(l)	scan	p2(b)
3	boot	appl	scan	p1(r)	p2(b)	p4	p3(l)	p5
4	p1(r)	p2(b)	p5	p3(l)	boot	appl	scan	p4
5	boot	p3(b)	p5	p1(r)	appl	p4	p2(l)	scan
6	p4	boot	p2(b)	p5	appl	p1(l)	scan	p3(r)
7	boot	appl	scan	p5	p3(l)	p4	p2(b)	p1(r)
8	p5	boot	p2(l)	p4	p3(r)	appl	scan	p1(b)

boot and events p1,...,p5. The resulting array consists of 18 rows.

The first thing to note is that using ASP enabled us to easily embed the additional requirements directly in the ASP program rather than employing an ad hoc and mostly manual approach. Furthermore, using ASP significantly reduced the size of the resulting SCA by 55.56% (cf. Table III).

C. Adding Attributes to Events

The next problem elaboration that we consider is related to the way the peripherals are connected to the laptop. Devices p1, p2, and p3 have to be connected to USB ports. Three ports are available: left, right, and back. In each test sequence, one port has to be assigned to a USB device.

1) Encoding: Predicate port(N,X,Y) states that USB device X is connected to port Y in row N of the array. This assignment should satisfy the following coverage criteria: (i) each USB device has to be connected to each port at least once and (ii) connections to the ports after the boot event should be made in any possible order. The above requirements can be formalised using few further rules.

In the following rules, we first specify the USB ports and devices. Then, it is expressed that each USB device is assigned to precisely one port in each test sequence. Finally, USB devices must not be connected to the same port in any sequence.

```
usbPort(right; left; back).
usbDevice(p1; p2; p3).
1{port(N,X,Y):usbPort(Y)}1 :- row(N),
                              usbDevice(X).
:- port(N,X,Y), port(N,Z,Y), X != Z.
```

Next, we state coverage criterion (i):

```
portCov(X,Y) :- port(N,X,Y).
:- usbDevice(X),usbPort(Y),not portCov(X,Y).
```

Lastly, we add rules for coverage criterion (ii):

```
portSeq(X,Y,Z) :- usbPort(X;Y;Z),
                  X!=Y,X!=Z,Y!=Z.
seqCov(N,X,Y,Z):-hb(N,boot,X),hb(N,X,Y),
                 hb(N,Y,Z).
pSeqCov(R,S,T) :- seqCov(N,X,Y,Z),
       port(N,X,R), port(N,Y,S), port(N,Z,T).
:- portSeq(X,Y,Z), not pSeqCov(X,Y,Z).
```

Let us denote the resulting program by $\Pi_3^3(n)$.

2) Discussion: Note that the additional conditions regarding the USB ports do not result in larger SCAs, still SCAs of size 8 can be obtained by computing the answer sets of $\Pi_3^3(8)$. Clearly, 8 is also an optimal bound. The runtime of the ASP solver is not affected by the additional requirements.

Kuhn et al. deal with the issue of USB ports by adding respective port assignments in a post-processing step once an SCA is computed. However, they do not provide details on which basis this is done, i.e., it is not clear if or in what sense they strive for systematic coverage.

D. Expressing Preferences

Any answer set of $\Pi_3^3(n)$ represents one admissible test plan for the application under test. Although each such SCA satisfies all of the requirements discussed so far, different SCAs could differ in their fault detection potential.

We next augment program $\Pi_3^3(n)$ by rules that state a preference relation among solutions, similar to program $\Pi_{grdy}^3(\cdot,\cdot)$ from the previous section. In particular, although any SCA guarantees full 3-way interaction coverage for some specified events, the degree of 4-way coverage of events may differ from one SCA to another. We will use the number of covered 4-sequences as discrimination criterion regarding the quality of solutions and consequently prefer SCAs that cover more 4-sequences over SCAs that cover fewer.

1) Encoding: We define program $\Pi_4^3(n)$ as $\Pi_3^3(n)$ augmented by the following rules:

```
covered(W,X,Y,Z) :- hb(N,W,X),hb(N,X,Y),
                    hb(N,Y,Z).
#maximize[covered(_,_,_,_)].
```

The first rule defines which 4-sequences are covered, the second rule states that the number of covered 4-sequences should be maximised.

2) Discussion: An SCA of size 8 corresponding to an answer set of $\Pi_4^3(8)$ is given in Table III. In the computation of the SCA, clasp has been configured to optimise a solution until no improvements can be found for 15 minutes.

On the other hand, Kuhn et al. has not handled preferences over solutions at all. The algorithm of Kuhn et al. is tailored for computing a single SCA. Thus, it may be hard to use such an algorithm to directly deal with optimisation issues, since this requires that solutions should be efficiently enumerated.

This case study demonstrates that often generic SCAs cannot be used in a real world scenario without significant modifications. In general, such modifications lead to a considerable overhead or are not feasible at all. By using ASP, however, a test author has a tool to state different requirements relevant for individual scenarios. Often, this will need only little effort such as adding few rules.

V. Related Work

Since the approach of Kuhn et al. [3], [4] is based on a greedy algorithm for generating SCAs, which have to be modified in a post-processing step to meet different user requirements, the ASP-based approach introduced in this paper is the first account of an approach for directly generating SCAs in the presence of expressible constraints and problem elaborations.

Closely related to our work are techniques for computing covering arrays (CAs), which we will review next. There, greedy algorithms that construct one row at a time are quite common. The most prominent representative is the AETG system [16]. Our greedy approach to compute SCAs is close in spirit to AETG-like algorithms since it also proceeds row by row. Also, meta-heuristics, like simulated annealing, tabu search, or genetic algorithms, have been applied for constructing CAs [17], [18], cf. respective overview articles [1], [2]. However, neither greedy techniques nor meta-heuristics can guarantee optimal bounds.

As a complete method being able to establish optimality of arrays, different SAT encodings have been considered [19], [20]. A distinctive feature of ASP compared to SAT is the high-level modelling capabilities of ASP that allow to model problems concisely at the first-order level as demonstrated by our SCA encodings. SAT is certainly a promising approach for tackling problems described in Section III, i.e., for computing SCAs and checking optimality of upper bounds. However, the problem variations discussed in Section IV require a formalism that allows for elaboration-tolerant representations, which is not a characteristic feature of SAT. Regarding modelling, it is to mention that Hnich et al. [19] and Banbara et al. [20] initially considered constrained programming (CP) models, which are subsequently translated to SAT. Though this has not been considered, further constraints, at least forbidden tuples, could be incorporated rather easily into the CP model. A comparison of ASP and constrained (logic) programming (CLP) is given in a related article [21]. There, the authors conclude that ASP allows for more declarative and concise problem representation and is easier to learn for newcomers than CLP. We also mention in passing that a vital aspect of the CP models was related to breaking symmetries, which obscures problem representation somewhat. Though symmetry breaking is also an issue in ASP, we experienced that adding symmetry-breaking constraints to our ASP programs has a quite negative effect on the performance of ASP solvers for improving upper bounds.

Cohen, Dwyer, and Shi [22], [23] introduced approaches that integrate techniques for generating covering arrays with SAT to deal with constraints. Forbidden tuples are represented as Boolean formulas and a SAT solver is used to compute models. They integrated SAT with greedy AETG-style algorithms and also with simulated annealing. Hence, their approach is closely related to our integration of ASP into a greedy procedure. Calvagna and Gargantini [24] follow a similar approach but they use an SMT solver instead of a SAT solver, which offers a richer language than plain SAT solvers. In their approach, constraints are stated as formal predicate expressions. Besides SMT, Calvagna and Gargantini also considered a model checker for verifying test predicates.

Bryce and Colbourn [25] distinguish forbidden tuples and tuples that should be avoided. They refer to the latter as soft constraints and they present an algorithm for generating CAs that avoids the violation of soft constraints. However, their algorithm cannot guarantee that certain tuples are avoided, hence it cannot deal with forbidden tuples or other hard constraints. Using ASP, soft constraints can be easily expressed by means of minimise or maximise statements. We illustrated in the previous section how one can combine hard integrity constraints with soft constraints to express that uncovered 4-sequences should be avoided.

VI. Conclusion and Future Work

In this paper, we dealt with the generation of SCAs, which have recently been advocated as suitable combinatorial design for event sequence testing [3], [4]. In particular, we applied ASP as a declarative approach for generating SCAs. While the only previously introduced algorithm is an AETG-like greedy algorithm [3], [4], ASP can be used as an exact method that combines high-level modelling capabilities with highly performative search engines [8].

To summarise, our contribution is two-fold: On the one hand, we introduced and showed feasibility of a new approach for generating SCAs that can be readily used as it is. On the other hand, we regard this work as a contribution towards methodology. While ASP is well established in other communities as a method to address problems from the area of artificial intelligence and knowledge representation, too little is known about ASP in the software-engineering community. Hence, we want to promote ASP as an approach to tackle challenging problems in the realm of combinatorial testing. Besides improving the state-of-the-art of event sequence testing, our aim is to show that ASP provides a tool that enables a tester to rapidly specify problems and to experiment with different formulations at a purely declarative level. ASP solvers are then used for computing solutions without the need of post-processing steps or developing dedicated algorithms.

For future work, we plan to deal with versions of SCAs for different testing applications like testing of concurrent programs where the order of shared variable accesses was

identified as crucial for triggering certain bugs that are otherwise hard to evoke [6], [26].

ACKNOWLEDGMENT

This work was supported by the Austrian Science Fund (FWF) under project P21698. We also would like to thank D. Richard Kuhn for providing us with related work [4].

REFERENCES

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Software Testing, Verification & Reliability*, vol. 15, no. 3, pp. 167–199, 2005.

[2] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv*, vol. 43, no. 2, pp. 11:1–11:29, 2011.

[3] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," NIST National Institute of Standards and Technology, NIST Special Publication 800–142, October 2010.

[4] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," 2010, submitted for publication. Available at http://csrc.nist.gov/groups/SNS/acts/documents/event-seq101008.pdf.

[5] M. J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," in *Proceedings of the 8th International Conference on Software Maintenance (ICSM 1992)*. IEEE Computer Society Press, 1992, pp. 272–281.

[6] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2007, pp. 533–536.

[7] C. Baral, *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, 2003.

[8] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński, "The second answer set programming competition," in *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, ser. LNCS, vol. 5753. Springer, 2009, pp. 637–654.

[9] "Combinatorial testing for event sequences," http://csrc.nist.gov/groups/SNS/acts/sequence_cov_arrays.html, last visited: July 18, 2011.

[10] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set solving," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*. AAAI Press/MIT Press, 2007, pp. 386–392.

[11] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *Proceedings of the 5th Logic Programming Symposium*, MIT Press, 1988, pp. 1070–1080.

[12] P. Simons, I. Niemelä, and T. Soininen, "Extending and implementing the stable model semantics," *Artificial Intelligence*, vol. 138, no. 1–2, pp. 181–234, 2002.

[13] M. Gebser, T. Schaub, and S. Thiele, "Gringo: A new grounder for answer set programming," in *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, ser. LNCS, vol. 4483. Springer, 2007, pp. 266–271.

[14] "Potassco—the potsdam answer set solving collection," http://potassco.sourceforge.net, last visited: July 18, 2011.

[15] http://www.kr.tuwien.ac.at/research/projects/mmdasp/collection-of-scas.tar.gz, last visited: July 18, 2011.

[16] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Software Eng.*, vol. 23, no. 7, pp. 437–444, 1997.

[17] M. B. Cohen, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, 2003, pp. 38–48.

[18] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, no. 1-2, pp. 143–152, 2004.

[19] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2-3, pp. 199–219, 2006.

[20] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, "Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers," in *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*, ser. LNCS, vol. 6397. Springer, 2010, pp. 112–126.

[21] A. Dovier, A. Formisano, and E. Pontelli, "An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems," *J. Exp. Theor. Artif. Intell.*, vol. 21, no. 2, pp. 79–121, 2009.

[22] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 16th ACM/SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2007, pp. 129–139.

[23] ——, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Software Eng.*, vol. 34, no. 5, pp. 633–650, 2008.

[24] A. Calvagna and A. Gargantini, "A formal logic approach to constrained combinatorial testing," *Journal of Automated Reasoning*, vol. 45, no. 4, pp. 331–358, 2010.

[25] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information & Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[26] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2008, pp. 329–339.