Event-Sequence Testing using Answer-Set Programming

Martin Brain[1], Esra Erdem[2], Katsumi Inoue[3], Johannes Oetsch[4], Jörg Pührer[4], Hans Tompits[4], and Cemal Yilmaz[2]

[1] *University of Oxford, Department of Computer Science,*
Oxford, OX1 3QD, UK
Email: martin.brain@cs.ox.ac.uk
[2]*Sabanci University, Faculty of Engineering and Natural Sciences,*
Orhanli, Tuzla, Istanbul 34956, Turkey
Email: {esraerdem,cyilmaz}@sabanciuniv.edu
[3]*National Institute of Informatics,*
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
Email: inoue@nii.ac.jp
[4]*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. In fact, many bugs are caused by the interaction of only a low number of such events. Based on this assumption, *sequence covering arrays* (SCAs) have recently been proposed 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 article, 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. Employing ASP for computing SCAs is further justified by new complexity results related to event-sequence testing that are established in this work.**

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

I. Introduction

This article is an extension of a previous conference version [1]. Besides an extended discussion of related work, the first important extensions is a complexity analysis of the main computational problem which gives further justification of our solution approach. The second important extension is that we use a different problem encoding which is, on the one hand, simpler than the one used in the previous conference paper, and, on the the other hand, significantly improves many results.

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, testing costs can be reduced drastically without sacrificing much fault-detection potential by using suitable combinatorial designs [2], [3]. To this end, Kuhn et al. [4] introduced *sequence covering arrays* (SCAs) for combinatorial event sequence testing. An SCA of strength t is an array of permutations of events such that every ordering of any t events appears as a subsequence of at least one row. For illustration, the following matrix is an SCA for four events {1,2,3,4} with $t = 3$:

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

Any ordering of three events can be found as subsequence of one row. If three particular events occur as subsequence of a row, we say that a row covers the three events. For example $(1, 2, 3)$ is a subsequence of the fourth row, $(1, 3, 2)$ is a subsequence of the second row, $(2, 3, 4)$ is covered by the third row, and so on and so forth.

SCAs are relevant for testing applications where the order of events is decisive. Examples of respective event sequences in such applications are user actions for user-interface testing, visited web pages in dynamic web applications, method calls for unit-testing in object-oriented programming, and shared variable accesses in multi-threaded programs as we already mentioned. If an SCA of strength t is used as basis for a test plan for such applications, i.e., each row of the SCA is turned into the specification of a test run that imposes a particular order on relevant event, not all permutations of events will be tested in general, but at least we have the guarantee that the potential interaction of any t events will be tested at least once.

In practice, a direct application of SCAs for testing is often impaired by additional constraints on the order of events. It can be necessary to exclude, for example, that a "paste" event happens before a "copy" event when testing a user interface. 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]. Hence, quite expressive constraints and variations from standard SCAs have to be taken into account. Furthermore, sometimes certain orderings are regarded as redundant and should be avoided to reduce testing costs. For example, the order in which devices are connected to a computer is not relevant if the computer is not booted.

One approach to address such considerations is to accordingly modify precomputed SCAs as exemplified by Kuhn et al. [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—which is a daily aspect in real-world system development—or one wants to experiment with different designs.

We propose to use *answer-set programming* (ASP) [7]–[9] 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 [10], [11]. Since it requires only little effort to state quite complex coverage conditions in ASP, a tester is able to rapidly specify and to experiment with different versions of SCAs.

This paper is organised as follows. In Section II, we review SCAs and ASP. In Section III, we analyse the intrinsic problem complexity of SCA computation which indeed shows that ASP is a suitable computational means. Then, in Section IV, we show how SCAs can be generated using ASP. 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 V, we turn towards a real-world example described by Kuhn et al. [4]. We discuss how the basic ASP encoding from Section IV can be refined, step-by-step, to take different constraints and problem variations into account. The resulting array is significantly smaller than the one of Kuhn et al. that was created by modifying a precomputed SCA. In fact, we show that our solution is optimal with respect to the specified coverage criteria. Finally, we discuss related work in Section VI and conclude in Section VII.

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. [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 sequence of t pairwise distinct elements of S. Following Kuhn et al. [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.

Definition 2: The *sequence covering array number for S and t*, $\mathrm{SCAN}(S, t)$, is the smallest n such that an (n, S, t)-SCA exists.

An (n, S, t)-SCA is *optimal* if $\mathrm{SCAN}(S, t) = n$. As usual, l is a *lower bound* for $\mathrm{SCAN}(S, t)$ if $l \leq \mathrm{SCAN}(S, t)$, and u is an *upper bound* for $\mathrm{SCAN}(S, t)$ if $\mathrm{SCAN}(S, t) \leq u$. 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 seven 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, and $(3,4,5)$ is covered by the fourth row of M (as well, $(3,4,5)$ is covered by the last 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 is available online [12]. These SCAs were computed using a simple greedy algorithm introduced by Kuhn et al. [4]. To compute a t-strength SCA for a set S of events, this algorithm iteratively computes single rows of the SCA: In each iteration, 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.

Recently, this algorithm has been extended to deal with forbidden orderings of events. In particular, if event x must not occur before event y in any test case, it is possible to specify the pair (x,y) as additional input of the algorithm. Subsequently, only rows that do not contain (x,y) as subsequence are added. Also, if a constraint is specified, the heuristic to add rows in reverse order is disabled.

Though the greedy algorithm can take simple constraints into account, one downside is that more complex constraints or other variations from plain SCAs 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]–[9] 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" [13], correspond to the solutions, and to find the answer sets for that program using an ASP solver. Due to the expressiveness of ASP, allowing, e.g., to represent aggregates and recursive definitions, and due to the continuous improvements of the efficiency of ASP solvers, such as `clasp` [14], 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 [15], systems biology [16], planning [17], diagnosis [18], [19], configuration [20], multi-agent systems [21], cladistics [22], [23], game content generation [24], and superoptimisation [25]. For a comprehensive introduction to ASP, we refer to the textbook by Baral [9].

In what follows, we recapitulate the basic elements of ASP. An *answer-set program* is a finite set of rules of the form

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

where $n \geq m \geq 0$, a_0 (called the *head* of the rule) is a propositional atom or $\bot$, $a_1,\ldots,a_n$ are propositional atoms, and the symbol "not" denotes *default negation*. The sequence $a_1,\ldots,a_m, \text{not } a_{m+1},\ldots,\text{not } a_n$ comprises the *body* of the rule. 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 [26]. 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 $a_0 \in I$. We say that 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

$$\begin{aligned} p &:- \; \text{not } q, \\ q &:- \; \text{not } p \end{aligned} \qquad (2)$$

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 [27]. Programs using these constructs can be viewed as abbreviations for particular normal programs [28]. 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 $p_i :-$ not p_{i+1}, $(1 \leq i \leq 7)$ as follows:

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

ASP solvers compute an answer set for a given program that contains variables after "grounding" the program, e.g., by the grounder `gringo` [29]. 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 $1\{p(i) : \mathit{index}(i)\}1$.

In addition to the constructs above, current state-of-the-art ASP solvers support many further 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 [30]. Note that, at term positions, upper-case letters denote variables, while lower-case letters denote constant symbols.

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. `clasp` supports maximise and minimise statements that allow the 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. Prelude: Complexity of SCA Generation

Deciding whether a logic program has an answer-set is NP-complete, thus computing answer-sets can be quite expensive. Indeed, the runtime of ASP solvers is exponential with respect to the number of atoms in the worst case. In this section, we analyse the computational worst-case complexity of generating SCAs. We assume that the reader is familiar with the basic concepts of complexity theory. For more information about complexity theory, we refer to the reference textbook by Papadimitriou [31].

For our complexity analysis, we actually study a slight generalisation of the problem of generating SCAs. On the one hand, usually not all permutations of events are allowed for testing, some could be excluded for various reasons. On the other hand, usually not all t-sequences need to be covered, some may be forbidden or regarded as redundant. We next formalise this natural generalisation as a decision problem and study its complexity.

Definition 3: An instance of the *generalised event sequence testing* (GEST) *problem* is a tuple (S, P, T, k), where P be is a set of permutations of a set S of symbols, T is a set of t-sequences over S with $t \geq 2$, and k is a positive integer. A tuple (S, P, T, k) is a yes-instance of GEST iff there exists a matrix M with at most k rows such that

(i) each row of M is an element from P, and
(ii) for each t-sequence $\sigma = (s_1, s_2, \ldots, s_t)$ from T, there is at least one row $\varrho = (a_{i1}, \ldots, a_{i|S|})$ in M such that σ is a subsequence of ϱ.

Theorem 1: GEST is NP-complete.

Proof:

Membership. We first show that GEST is in NP. Any instance (S, P, T, k) of GEST can be decided by non-deterministically "guessing" a $k \times |S|$ matrix M of symbols from S and checking conditions (i) and (ii) from Definition 3 in polynomial time.

Hardness. To show NP-hardness, we reduce the NP-hard problem of checking set coverage to GEST. Formally, an instance of *set cover* (SC) is a tuple (V, F, k), where V is a set of elements, F is a collection of subsets of V, and k is a positive integer. A tuple (V, F, k) is a yes-instance of SC iff there is a subcollection $F' \subseteq F$ of size at most k whose union contains each element of V. It is well known that SC is NP-complete [32].

Let (V, F, k) be an instance of SC. Assume that "$\square$" is a separation symbol not contained in V. Define

$$S = V \cup \{\square\}\,.$$

For each $f \in F$, construct a permutation π_f of S by arbitrarily arranging the symbols from f before $\square$ and the symbols in $V \setminus f$ after $\square$. Define

$$P = \{\pi_f \mid f \in F\}$$

and

$$T = \{(v, \square) \mid v \in V\}\,.$$

Note that $|P| = |F|$. We show that (V, F, k) is a yes-instance of SC iff (S, P, T, k) is a yes-instance of GEST.

Assume that (V, F, k) is a yes-instance of SC. Hence, there exists a set $F' \subseteq F$ of size at most k whose union contains each element of V. Construct a matrix M such that $\pi_f \in P$ is a row of M iff $f \in F'$. Clearly, M is a matrix from symbols from S that satisfies condition (i) of Definition 3. We show that M satisfies condition (ii) as well. Towards a contradiction, assume that there is a 2-sequence $(v, \square)$ in T and there is no row in M such that v occurs before $\square$. Since (V, F, k) is a yes-instance of SC, F' contains at least one set f with $v \in f$. Since π_f is a row of M, and v occurs before $\square$ in π_f by construction, we arrive at a contradiction. Hence, (S, P, T, k) is a yes-instance of GEST.

For the converse, assume that (S, P, T, k) is a yes-instance of GEST. Hence, there exists a matrix M that satisfies conditions (i) and (ii) from Definition 3. We show that then (V, F, k) is a yes-instance of SC. Define set F' as a subset of F that contains an element $f \in F$ iff $(*)$ there is a row π of M such that all elements of f occur before $\square$ in π. Clearly, F' is of size at most k since M consists of at most k rows, and, for any row of M, precisely one $f \in F$ satisfies $(*)$. It remains to show that for each $v \in V$, v is contained in some set in F'. Towards a contradiction, assume that for some $v \in V$ there is no set in F' that contains v. Since (S, P, T, k) is a yes-instance of GEST, and $(v, \square) \in T$, it follows that in one row π_f of M, v occurs before $\square$. By construction of F', F' contains a set $f \in F$ consisting of all symbols of π_f that occur before $\square$. Hence, $v \in f$ which contradicts the assumption that no such set in F exists. So, (V, F, k) must be a yes-instance of SC. ■

Hence, any approach that is capable of deciding problems in GEST cannot avoid worst-case exponential runtime behaviour unless P $=$ NP. Note that the SCA generation problems studied in this paper are instances of GEST. Moreover, for any problem in NP, there exists a uniform ASP encoding [33], [34]. Hence, NP-completeness of GEST further justifies ASP as a tool to formalise and compute such problems.

Although GEST is NP-complete, one could further ask whether GEST is *fixed-parameter tractable* for a suitable problem parameter. Roughly speaking, fixed-parameter tractability means that a problem can be solved efficiently, i.e., in polynomial time, for fixed values of the parameter; details on parameterised complexity can be found elsewhere [35], [36]. A natural choice for such a parameter for a problem instance (S, P, T, k) would be the size k of the SCA because it can be assumed to be small in practice. We denote the parameterised version of GEST with k as parameter as the *standard parameterisation of* GEST.

In more formal terms, a parameterisation of a decision problem is obtained by assigning a natural number to each problem instance. A parameterised decision problem is fixed-parameter tractable if a problem instance x with parameter k can be decided in running time $f(k) \cdot |x|^{O(1)}$, where f is a computable function which is independent of $|x|$.

In standard complexity theory, problems are classified and ordered into hierarchies using polynomial-time reductions. Under parameterised complexity, parameterised reductions, so-called *fpt-reductions*, are used for this purpose. An fpt-reduction from a parameterised problem P to a parameterised problem Q is a function ϕ such that for any instance x of P

(i) $\phi(x)$ can be computed in time $f(k) \cdot |x|^{O(1)}$, where k is the parameter of x,

(ii) $\phi(x)$ is a yes-instance of Q iff x is a yes-instance of P, and

(iii) if k is the parameter of x and k' is the parameter of $\phi(x)$, then $k' \leq g(k)$, for some computable function g.

The class FPT contains all fixed-parameter tractable problems. Note that FPT is closed under fpt-reductions. Similar to the polynomial hierarchy in standard complexity theory, a hierarchy of classes W[i] has been introduced with FPT at its lowest level. In particular, FPT $=$ W[0] and W[i] $\subseteq$ W[j], for all $i \leq j$. All classes W[i] are closed under fpt-reductions. Moreover, analogous to P $\subseteq$ NP, it is not known whether the inclusions W[i] $\subseteq$ W[j] are proper, but most experts believe this to be the case. The following result implies that GEST is not in FPT unless the W-hierarchy collapses up to the second level.

Theorem 2: The standard parameterisation of GEST is W[2]-complete.

Proof:

Consider an instance (V, F, k) of SC. If we take k, i.e., the size of the subcollection $F' \subseteq F$ whose union contains each element of V, as parameter, SC is W[2]-complete [35].

Membership. To show membership in W[2], we use an fpt-reduction from GEST to SC. Let (S, P, T, k) be an instance of GEST. For any $\pi \in P$, construct a set $f_\pi \subseteq T$ that contains any t-sequence $\tau \in T$ iff the elements of τ occur in π in the same order as in τ. Define

$$F = \{f_\pi \mid \pi \in P\}\,.$$

We show that (S, P, T, k) is a yes-instance of GEST iff (T, F, k) is a yes-instance of SC.

Assume that (S, P, T, k) is a yes-instance of GEST. Hence, there exists a matrix M satisfying conditions (i) and (ii) from Definition 3. Define F' as the subset of F that contains f_π iff π is a row of M. Clearly, the size of F' is at most k. It remains to show that the union of the elements of F' contain each $\tau \in T$. Towards a contradiction, assume that there is an element $\tau \in T$ such that no set in F' contains τ. Since (S, P, T, k) is a yes-instance of GEST, M contains a row $\pi \in P$ such that the symbols in τ occur in π in the same order as in τ. Thus $\tau \in f_\pi$. Since $f_\pi \in F'$, we arrive at a contradiction. Therefore, (T, F, k) must be a yes-instance of SC.

Assume now that (T, F, k) is a yes-instance of SC. Hence, there is a subset $F' \subseteq F$ of size at most k whose union contains each element of T. Construct a matrix M according to Definition 3 such that M contains, for each $f \in F'$, one row $\pi \in P$ that satisfies $f_\pi = f$. Clearly, M consists of at most k rows from P. It remains to show that M satisfies condition (ii) of Definition 3. Towards a contradiction, assume that there is a t-sequence $\tau \in T$ such that no row contains the symbols in τ in the same order as in τ. Since (T, F, k) is a yes-instance of SC, there is a set $f \in F'$ that contains τ. Since M contains a row π with $f_\pi = f$, it follows from the construction of f_π that π contains the symbols in τ in the same order as in τ. We thus arrive at a contradiction, and so (S, P, T, k) is yes-instance of GEST.

Hardness. To show that GEST is W[2]-hard, we define an fpt-reduction from SC to GEST. In fact, the reduction used in the hardness proof of Theorem 1 is such an fpt-reduction since the problem parameter k is preserved. Hence, W[2]-hardness of GEST follows. ■

Hence, even if we are interested only in relatively small test plans, it is presumably not possible to avoid worst-case exponential runtime.

IV. 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. We in particular demonstrate that (i) the declarative nature of ASP encodings can help to state complex coverage criteria, involving constraints and possibly recursive definitions with ease in a concise and elaboration-tolerant way, and (ii) when the declarative nature of ASP encodings is coupled with ever-improving efficiency of ASP solvers, even simple encodings that closely reflect the problem statement in natural language can provide better SCAs (e.g., smaller SCAs) compared to those obtained from a dedicated algorithm.

Ahead of our discussion in Section V addressing how different problem elaborations can be incorporated into a single answer-set program, we introduce in what follows an answer-set program for computing generic SCAs. This program will serve as basis for further problem elaboration discussed in the sequel. We also introduce a new greedy approach that combines a simple variation of the basic ASP encoding with an iterative greedy procedure.

```
% ASP encoding for (n,s,3)-SCAs
sym(1..s). row(1..n).

% guess happens-before relation
1{hb(N,X,Y),hb(N,Y,X)}1 :- row(N),
                           sym(X;Y), X != Y.

% happens-before is transitive
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).

% happens-before is irreflexive
:- hb(N,X,X).

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

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

A. Basic Encoding

To begin with, we present an ASP program for computing (n, s, t)-SCAs with $t = 3$. We assume throughout that $s \geq 3$. Note that this program can be changed in a straightforward way to obtain encodings for any fixed $t > 3$. An encoding for SCAs where t is not fixed can be obtained using *disjunctive ASP* [13]—this is, however, beyond the scope of this article. For the sake of understandability, we introduce our encoding step-by-step.

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 `hb(N,X,Y)` expressing that in row `N` event symbol `X` happens before symbol `Y`. The basic idea is that we will define this happens-before relation in a way that it is, for each row, a strict total order on the event symbols.

The first rule states that for any two distinct symbols `X` and `Y` in each row, either `X` happens before `Y` or `Y` happens before `X`:

```
1 {hb(N,X,Y),hb(N,Y,X)} 1 :- row(N),
                             sym(X;Y), X != Y.
```

We need further rules to guarantee that the happens-before relation is indeed a strict total order. In particular, we need rules that guarantee that the happens-before relation is transitive and irreflexive. Now, transitivity can be expressed in a straightforward way:

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

Directly expressing inductive definitions as above is a particular strength of ASP and distinguishes it from related declarative approaches that are more based on the semantics of classical first-order logic.

To state that the happens-before relation is irreflexive, a simple additional constraint is required.

```
:- hb(N,X,X).
```

Hence, it is forbidden that a symbol occurs before itself.

This finally implies that the happens-before relation is a strict total order on the event symbols $\{1, \ldots, s\}$ which further implies that each row is a permutation of the event symbols when we order them according to the happens-before relation.

It only remains to require that each 3-sequence of symbols is covered by some row. Observe that a 3-sequence is a triple of pairwise distinct symbols. 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).
:- not covered(X,Y,Z),
   sym(X;Y;Z), X!=Y, Y!=Z, X!=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 4: An answer set X of $\Pi^3(n, s)$, for $s \geq 3$, *represents* an $n \times s$ matrix M iff, for any i, j, $1 \leq i < j \leq s$, and any r, $1 \leq r \leq n$, $M_{r,i} = s_1$ and $M_{r,j} = s_2$ precisely in case X contains the atom $\mathtt{hb}(r, s_1, s_2)$.

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

```
hb(1,1,4), hb(1,3,1), hb(1,2,3),
hb(1,5,2), hb(1,5,3), hb(1,2,1),
hb(1,3,4), hb(1,2,4), hb(1,5,4),
hb(1,5,1).
```

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).

Table I
UPPER BOUNDS n FOR SCAN$(s, 3)$ OBTAINED BY KUHN ET AL. [4] AND OUR ASP ENCODING. A STAR INDICATES AN OPTIMAL BOUND.

s	n (Kuhn et al. [4])	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	10
17	20	11
18	20	12
19	22	12
20	22	12
21	22	12
22	22	12
23	24	13
24	24	13
25	24	14
26	24	14
27	26	14
28	26	14
29	26	14
30	26	15
40	32	17
50	34	18
60	38	20
70	40	22
80	42	23

2) Discussion: Program $\Pi^3(n, s)$ nicely illustrates how challenging search problems can be concisely encoded using ASP: The program consists of only seven 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 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. [4] is given in Table I. Computation times for the reported upper bounds range from fractions of a second to 180 minutes. We have considered strength 3 SCAs for five to 80 events. The known upper bounds reported by Kuhn et al. [4] could be improved throughout. The more events are considered, the more drastic are the improvements; for some arrays, we need up to 46.88% less test sequences. Such savings are especially significant in settings where running single test sequences are costly.

For small SCAs—viz. for 5 to 10 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 $\mathrm{SCAN}(10, 3) = 9$, 9 is a trivial lower bound for any $\mathrm{SCAN}(s, 3)$ with $s > 10$. 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. Although 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. [4] seems to scale quite well; the authors report on SCAs for up to 80 events not only for strength 3 arrays, but they also consider arrays of strength 4 where our ASP approach quickly reaches its limits.

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. [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.

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:

```
hb(N,X,Y) :- row(N),hb(N-1,Y,X),
             N #mod 2 == 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 happens-before relation is the inverse of the happens-before 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. [4]. Next, we present an ASP-based greedy algorithm inspired by that of Kuhn et al. that does not rely on adding inverted rows.

```
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:     $n \Leftarrow n + 1$
5:     $X \Leftarrow$ answer set of $\Pi^3_{grdy}(s, n) \cup N$
6:     $N \Leftarrow N \cup X|_{\mathtt{hb}/3}$
7: until $N$ represents an $(n, s, 3)$-SCA
```

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

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 $\Pi^3_{grdy}(s, i)$, which is depicted in Figure 3. Program $\Pi^3_{grdy}(s, i)$ takes the number s of events and a row index i 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 $\Pi^3_{grdy}(s, i)$ 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).
#maximize[covered(_,_,_,_)].
```

Program $\Pi^3_{grdy}(s, i)$ is quite similar to $\Pi^3(n, s)$. However, each answer set of $\Pi^3_{grdy}(s, i)$ corresponds only to a single row with index i of an SCA. The idea is to represent preceding rows with index 1 to $i-1$ by means of facts `hb`/3. These facts are joined with $\Pi^3_{grdy}(s, i)$. Then, the answer sets of $\Pi^3_{grdy}(s, i)$ correspond to those rows that obtain maximal coverage of previously uncovered 3-sequences. The encoding follows the *guess, check, and optimise pattern*, i.e., 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 (cf. 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 `hb`/3 equals the empty set. In each iteration, $\Pi^3_{grdy}(s, i) \cup N$ is used to compute the next row of the SCA that obtains maximal increase of previously uncovered 3-sequences. The respective `hb`/3 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

```
% guess single row with index i of an
% (n,s,3)-SCA
sym(1..s).

% guess happens-before relation
1{hb(i,X,Y),hb(i,Y,X)}1 :- sym(X;Y), X != Y.

% happens-before is transitive
hb(i,X,Z) :- hb(i,X,Y), hb(i,Y,Z).

% happens-before is irreflexive
:- hb(i,X,X).

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

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

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. We used time limits of up to 10 minutes for computing single rows, depending on the problem size.

2) Discussion: To sum up our results so far, our analysis of the heuristic proposed by Kuhn et al. [4] using ASP has pinpointed some shortcomings of the former and has helped us to learn more about the problem at hand. Furthermore, we have proposed a new greedy algorithm making use of a slight variation of $\Pi^3(n,s)$. The ASP solver takes care entirely of the greedy optimisation of the single rows of the SCA. The algorithm thus only keeps track of the partial SCA and incrementally calls the ASP solver to compute new rows.

Table II summarises a comparison of our greedy ASP algorithm with the greedy algorithm of Kuhn et al. [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 bounds for strength 3 SCAs for up to 80 symbols and can also improve bounds for strength 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.

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

s	$t=3$		$t=4$	
	Kuhn et al. [4]	ASP	Kuhn et al. [4]	ASP
10	14	11	66	55
20	22	17	120	104
30	26	22	156	149
40	32	26	182	181
50	34	29	204	-
60	38	32	222	-
70	40	35	238	-
80	42	36	250	-

V. 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. [4] for making our point. The specification of this testing problem is as follows: There are five 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 eight 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 eight events, optimal SCAs of strength 3 comprise eight 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 $\Pi^3_1(n)$.

2) Discussion: Recall that $(8,8,3)$-SCAs are optimal for eight symbols. Since, $\Pi^3_1(8)$ 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 $\Pi^3_1(9)$ 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 1 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 two-way coverage of the symbols from predicate `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 $\Pi^3_2(n)$.

2) Discussion: Program $\Pi^3_2(n)$ 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 $\Pi^3_2(7)$ 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 precomputed $(12,7,3)$-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 V-A) are replaced. However, this requires that further rows are added to preserve coverage. Note that this testing application was considered before the greedy algorithm was extended to directly express simple constraints [4]. 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 `boot` and events `p1`, ..., `p5`. The resulting array consists of 19 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 eleven rows (57.94%), 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

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)

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 three-way interaction coverage for some specified events, the degree of four-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. The complete ASP encoding $\Pi_4^3(8)$ is given in Figure 4.

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. [4] have 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.

VI. Related Work

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. We note, however, that after the previous conference version of this paper [1], our idea was picked up soon by Banbara et al. [37]. They proposed a constraint-programming encoding called the *incidence-matrix model* for generating SCAs. Although we could either reproduce or improve all bounds for $\mathrm{SCAN}(s, 3)$ reported by Banbara at al. [37] in this paper, the encoding based on the incidence-matrix model scales better than ours for $\mathrm{SCAN}(s, 4)$ SCAs.

Closely related to our work are techniques for computing covering arrays (CAs), which we will review next. An overview of different approaches and tools for generating CAs is given by Grindal, Offutt, and Andler [2]. There, greedy algorithms that construct one row at a time are quite common. The most prominent representative is the AETG system [38]. 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 [39], [40] (cf. respective overview articles for more details [2], [3]). Greedy algorithms usually scale well while meta-heuristics tend to produce arrays of smaller sizes [39]. 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 [41], [42]. Similar to our ASP encoding, SAT encodings allow to compute combinatorial designs as a whole. From a computational point of view, SAT and ASP are closely related and ASP solvers like `clasp` use many techniques also used by SAT solvers like conflict-driven clause learning. In fact, `clasp` can be used as a SAT solver itself—it even outperformed state-of-the-art SAT solvers at the SAT 2011 competition. 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 IV, i.e., for computing SCAs and checking optimality of upper bounds. However, the problem variations discussed in Section V 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. [41] and Banbara et al. [42] initially considered constraint programming (CP) models, which are subsequently translated to SAT. Although 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 constraint (logic) programming (CLP) is given in a related

```
% ASP encoding for the laptop example
sym(boot; p1; p2; p3; p4; p5; appl; scan).
row(1..n).

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

% guess happens-before relation
1{hb(N,X,Y),hb(N,Y,X)}1 :- row(N),
                           sym(X;Y), X != Y.
% happens-before is transitive
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).
% happens-before is irreflexive
:- hb(N,X,X).

% check three-way and two-way coverage
covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
:- not covered(X,Y,Z),
   threeWay(X;Y;Z), X!=Y, Y!=Z, X!=Z.
covered(X,Y) :- hb(_,X,Y).
:- twoWay(X;Y), X != Y, not covered(X,Y).

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

% coverage of USB ports
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.

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

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).

% maximise covered 4-sequences
covered(W,X,Y,Z) :- hb(N,W,X),hb(N,X,Y),
                    hb(N,Y,Z).
#maximize[covered(_,_,_,_)].
```

Figure 4. ASP encoding $\Pi^3_4(8)$.

article [43]. There, the authors conclude that ASP allows for more declarative and concise problem representations and is easier to learn for beginners than CLP.

The need for stating constraints and other user requirements in combinatorial interaction testing for real-world applications has been discussed by different authors [38], [44]–[50]. The prevalent approach is to first generate a CA and then to delete and permute rows that are not consistent with certain requirements. The number of rows that need to be replaced can be vast and this approach can lead to a considerable increase of the array size [49]. This applies not only for CAs but for SCAs as well as we have illustrated in the previous section. Another common method requires remodelling of the specification [38], [45].

The tool PICT [47], also based on an AETG-like greedy algorithm, allows to directly express constraints; however, the details how this is realised are not accessible.

Cohen, Dwyer, and Shi [48], [49] 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 [50] 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 which would also be suitable for specifications involving temporal constraints and state transitions.

Bryce and Colbourn [46] 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. Some ASP solvers, like DLV [51], allow to express soft constraints even more directly (in the case of DLV, in the form of *weak constraints*). 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. For even more fine-grained modelling, ASP allows to assign different priorities to soft constraints and maximise, resp., minimise, statements.

VII. Conclusion and Future Work

In this paper, we dealt with the generation of SCAs, which have recently been advocated as suitable combinatorial design concepts for event sequence testing [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 [4], ASP can be used as an exact method that combines high-level modelling capabilities involving recursive definitions, default negation, hard constraints, soft constraints, and aggregates with highly performative search engines [10], [11].

Our contributions can be summarised as follows:

- We established new complexity results related to computing SCAs. In particular, we showed that GEST is NP-complete and its standard parameterisation is W[2]-complete.
- We introduced a novel technique to use ASP for computing SCAs as a whole. The SCAs obtained using ASP are significantly smaller than those generated using the greedy algorithm of Kuhn et al. [4]. For some SCAs, optimality of upper bounds could be established.
- We integrated our ASP approach into a greedy algorithm that allows to compute SCAs in a one-row-at-a-time fashion. Hence, we obtain a more scalable algorithm without sacrificing the modelling capacities of ASP for specifying complex testing problems.
- We dealt with problem elaborations that are indispensable for testing real-world applications. In particular, we addressed how constraints and other application-specific requirements can be handled directly at the level of the ASP representation without a further need for post-processing steps.

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, there is too little awareness of 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], [52]. We want to address not only the problem of statically generating suitable designs, but we also want to do this in an *online fashion* where an ASP solver is coupled with a scheduler to improve coverage with respect to different interleaving metrics. Such an online approach would also allow to deal with, e.g., exceptional events in a more interactive testing environment. In the long term, we plan to develop support for a tester regarding modelling of a system's test space without requiring expert knowledge on ASP—a front-end language for ASP tailored to specific testing domains could be the right way of doing this.

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] E. Erdem, K. Inoue, J. Oetsch, J. Pührer, H. Tompits, and C. Yilmaz, "Answer-set programming as a new approach to event-sequence testing," in *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*. XPS, 2011, pp. 25–34.

[2] 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.

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

[4] D. R. Kuhn, J. M. Higdon, J. Lawrence, R. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012)*. IEEE, 2012, pp. 601–609.

[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] V. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm," in *The Logic Programming Paradigm: A 25-Year Perspective*. Springer, 1999, pp. 375–398.

[8] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3-4, pp. 241–273, 1999.

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

[10] 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. Lecture Notes in Computer Science, vol. 5753. Springer, 2009, pp. 637–654.

[11] F. Calimeri, G. Ianni, and F. Ricca, "The third open answer set programming competition," *CoRR*, vol. abs/1206.3111, 2012. [Online]. Available: http://arxiv.org/abs/1206.3111

[12] "Combinatorial testing for event sequences," http://csrc.nist.gov/groups/SNS/acts/sequence_cov_arrays.html, last visited: July 11, 2012.

[13] M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, vol. 9, pp. 365–385, 1991.

[14] 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.

[15] A. Polleres, "Semantic Web Languages and Semantic Web Services as Application Areas for Answer Set Programming," in *Nonmonotonic Reasoning, Answer Set Programming and Constraints*, 2005.

[16] S. Grell, T. Schaub, and J. Selbig, "Modelling biological networks by action languages via answer set programming," in *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, ser. Lecture Notes in Computer Science, vol. 4079. Springer, 2006, pp. 285–299.

[17] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, "Planning under incomplete knowledge," in *Proceedings of the First International Conference on Computational Logic (CL 2000)*, ser. Lecture Notes in Computer Science, vol. 1861. Springer, 2000, pp. 807–821.

[18] T. Eiter, W. Faber, N. Leone, and G. Pfeifer, "The diagnosis frontend of the DLV system," *AI Communications*, vol. 12, no. 1-2, pp. 99–111, 1999.

[19] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An A-prolog decision support system for the Space Shuttle," in *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL 2001)*, ser. Lecture Notes in Computer Science, vol. 1990. Springer, 2001, pp. 169–183.

[20] T. Soininen and I. Niemelä, "Developing a declarative rule language for applications in product configuration," in *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL 1999)*, ser. Lecture Notes in Computer Science. Springer, 1999.

[21] C. Baral and M. Gelfond, "Reasoning agents in dynamic domains," in *Logic-based Artificial Intelligence*. Kluwer Academic Publishers, 2000, pp. 257–279.

[22] E. Erdem, V. Lifschitz, and D. Ringe, "Temporal phylogenetic networks and logic programming," *Theory and Practice of Logic Programming*, vol. 6, no. 5, pp. 539–558, 2006.

[23] D. R. Brooks, E. Erdem, S. T. Erdogan, J. W. Minett, and D. Ringe, "Inferring phylogenetic trees using answer set programming," *Journal of Automated Reasoning*, vol. 39, no. 4, pp. 471–511, 2007.

[24] A. M. Smith and M. Mateas, “Answer set programming for procedural content generation: A design space approach,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011.

[25] M. Brain, T. Crick, M. De Vos, and J. Fitch, “TOAST: Applying answer set programming to superoptimisation,” in *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, ser. Lecture Notes in Computer Science. Springer, 2006.

[26] 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.

[27] 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.

[28] P. Ferraris and V. Lifschitz, “Mathematical foundations of answer set programming,” in *We Will Show Them! Essays in Honour of Dov Gabbay, Volume One*. College Publications, 2005, pp. 615–664.

[29] 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. Lecture Notes in Computer Science, vol. 4483. Springer, 2007, pp. 266–271.

[30] “Potassco—the Potsdam answer set solving collection,” http://potassco.sourceforge.net, last visited: July 11, 2012.

[31] C. H. Papadimitriou, *Computational Complexity*. Addison-Wesley, New York, 1994.

[32] R. Karp, “Reducibility among Combinatorial Problems,” in *Complexity of Computer Computations*. Plenum Press, 1972, pp. 85–103.

[33] J. S. Schlipf, “The expressive powers of the logic programming semantics,” *Journal of Computer and System Sciences*, vol. 51, no. 1, pp. 64–86, 1995.

[34] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, “Complexity and expressive power of logic programming,” *ACM Computing Surveys*, vol. 33, no. 3, pp. 374–425, 2001.

[35] R. Downey and M. R. Fellows, *Parameterized complexity*. New York: Springer, 1999.

[36] J. Flum and M. Grohe, *Parameterized Complexity Theory*, ser. Text in Theoretical Computer Science. Springer, 2006.

[37] M. Banbara, N. Tamura, and K. Inoue, “Generating event-sequence test cases by answer set programming with the incidence matrix,” in *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012), LIPIcs, Schloss Dagstuhl*, vol. 12, 2012, pp. 86–97.

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

[39] 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.

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

[41] 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.

[42] 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. Lecture Notes in Computer Science, vol. 6397. Springer, 2010, pp. 112–126.

[43] A. Dovier, A. Formisano, and E. Pontelli, “An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems,” *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 21, no. 2, pp. 79–121, 2009.

[44] A. Hartman and L. Raskin, “Problems and algorithms for covering arrays,” *Discrete Mathematics*, vol. 284, no. 1-3, pp. 149–156, 2004.

[45] C. M. Lott, A. Jain, and S. R. Dalal, “Modeling requirements for combinatorial software testing,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[46] 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.

[47] J. Czerwonka, “Pairwise testing in real world,” in *Proceedings of the 24th Pacific Northwest Software Quality Conference (PNSQC 2006)*, 2006, pp. 419–430.

[48] 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.

[49] ——, “Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[50] 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.

[51] “DLV system,” http://www.dlvsystem.com, last visited: July 11, 2012.

[52] 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.