catnap: Generating Test Suites of Constrained Combinatorial Testing with Answer Set Programming*

Mutsunori Banbara[1], Katsumi Inoue[2], Hiromasa Kaneyuki[1], Tenda Okimoto[1], Torsten Schaub[3], Takehide Soh[1], and Naoyuki Tamura[1]

[1] Kobe University, 1-1 Rokko-dai Nada-ku Kobe Hyogo, 657-8501, Japan
[2] NII, 2-1-2 Hitotsubashi Chiyoda-ku Tokyo, 101-8430, Japan
[3] Universität Potsdam, August-Bebel-Strasse 89, D-14482 Potsdam, Germany

Abstract. We develop an approach to test suite generation for Constrained Combinatorial Testing (CCT), one of the most widely studied combinatorial testing techniques, based on Answer Set Programming (ASP). The resulting *catnap* system accepts a CCT instance in fact format and combines it with a first-order encoding for generating test suites, which can subsequently be solved by any off-the-shelf ASP systems. We evaluate the effectiveness of our approach by empirically contrasting it to the best known bounds obtained via dedicated implementations.

1 Introduction

Software testing can generally be defined as the task of analyzing software systems to detect failures. Recently, software testing has become an area of increasing interest in several Software Engineering communities involving both researchers and practitioners, such as the international series of ICST/IWCT conferences. The typical topics of this area include *model-based testing*, *combinatorial testing*, *security testing*, *domain specific testing*, etc. In this paper, we consider a problem of generating test suites (viz. sets of test cases) for Combinatorial Testing (CT; [7, 15, 19]) and its extensions.

CT is an effective black-box testing technique to detect elusive failures of software. Modern software systems are highly configurable. It is often impractical to test all combinations of configuration options. CT techniques have been developed especially for such systems to avoid falling into combinatorial explosion. CT relies on the observation that most failures are caused by interactions between a small number of configuration options. For example, *strength-t CT* tests all t-tuples of configuration options in a systematic way. Such testing requires much smaller test suites than exhaustive testing, and is more effective than random testing.

* This work was partially funded by JSPS (KAKENHI 15K00099) and DFG (SCHA 550/11).

Constrained CT (*CCT*; [8]) is an extension of CT with constraint handling; it is one of the most widely studied CT techniques in recent years. In many configurable systems, there exist constraints between specific configuration options that make certain combinations invalid. CCT provides a combinatorial approach to testing, involving hard and soft constraints on configuration options. Hard constraints must be strictly satisfied. Soft constraints must not necessarily be satisfied but the overall number of violations should be minimal. Therefore, CCT cannot only exclude *invalid* tuples that cannot be executed, but also minimize the ones that might be undesirable.

The CCT problem of generating optimal (smallest) CCT test suites is known to be difficult. Several methods have been proposed such as greedy algorithms [8, 10, 21, 22], metaheuristics-based algorithms [12, 14], and Satisfiability Testing (SAT) [16, 20]. However, each method has strengths and weaknesses. Greedy algorithms can quickly generate test suites. Metaheuristics-based implementations can give smaller ones by spending more time. On the other hand, both of them cannot guarantee their optimality. Although complete methods like SAT can guarantee optimality, it is costly to implement a dedicated encoder from CCT problems into SAT. For constraint solving in CCT, most methods utilize off-the-shelf constraint solvers as back-ends [4]. They are used to check whether each test case satisfies a given set of constrains during test suite generation as well as to calculate valid tuples at prepossessing. However, there are few implementations that provide CCT with soft constraints and limiting resources (the number of test cases, time, and etc). It is therefore challenging to develop a universal CCT solver which can efficiently generate optimal test suites as well as suboptimal ones for CCT instances having a wide range of hard and soft constraints, even if the resources are limited.

In this paper, we describe an approach to solving the CCT problems based on Answer Set Programming (ASP; [4]). The resulting *catnap* system accepts a CCT instance in fact format and combines it with a first-order ASP encoding for CCT solving, which is subsequently solved by an off-the-shelf ASP system, in our case *clingo*. Our approach draws upon distinct advantages of the high-level approach of ASP, such as expressive language, extensible encodings, flexible multi-criteria optimization, etc. However, the question is whether *catnap*'s high-level approach matches the performance of state-of-the-art CCT solving techniques. We address this question by empirically contrasting *catnap* with dedicated implementations. From an ASP perspective, we gain insights into advanced modeling techniques for CCT. *catnap*'s encoding for CCT solving has the following features: (a) a series of compact ASP encodings, (b) easy extension for CCT under limiting resources, and (c) easy extension of CCT with soft constraints.

In the sequel, we assume some familiarity with ASP, its semantics as well as its basic language constructs. Our encodings are given in the language of *gringo* series 4. Although we provide a brief introduction to CCT in the next section, we refer the reader to the literature [17] for a broader perspective.

[4] SAT solvers in [8, 12, 14, 16, 20], PB (Pseudo Boolean) solver in [22], and CSP (Constraint Satisfaction Problem) solver in [21].

	Product Line Options				
	Display	Email Viewer (Email)	Camera	Video Camera (Video)	Video Ringtones (Ringtones)
Possible Values	16 Million Colors (16MC)	Graphical (GV)	2 Megapixels (2MP)	Yes	Yes
	8 Million Colors (8MC)	Text (TV)	1 Megapixel (1MP)	No	No
	Black and White (BW)	None	None		

Constraints on valid configurations:

(1) Graphical email viewer **requires** color display.
(2) 2 Megapixel camera **requires** a color display.
(3) Graphical email viewer **not supported** with the 2 Megapixel camera.
(4) 8 Million color display **does not support** a 2 Megapixel camera.
(5) Video camera **requires** a camera **and** a color display.
(6) Video ringtones **cannot occur** with No video camera.
(7) The combination of 16 Million colors, Text and 2 Megapixel camera is **not supported**.

Fig. 1. Mobile phone product line in [8]

2 Background

Generating a CCT test suite is to find a *Constrained Mixed-level Covering Array* (*CMCA*; [8]), which is an extension of a *Covering Array* [9] with constraints. A *CMCA* of size N is a $N \times k$ array $A = (a_{ij})$, written as $CMCA(t, k, (v_1, \ldots, v_k), \mathcal{C})$. The integer constant t is the *strength* of the coverage of interactions, k is the number of parameters, and v_j is the number of values for each parameter j ($1 \leq j \leq k$). The constraint $\mathcal{C}$ is given as a Boolean formula in Conjunctive Normal Form (CNF):

$$\mathcal{C} = \bigwedge_{1 \leq \ell \leq h} \left(\bigvee_{1 \leq m \leq m_\ell} pos(j_m^\ell, d_m^\ell) \quad \vee \bigvee_{m_\ell + 1 \leq n \leq n_\ell} neg(j_n^\ell, d_n^\ell) \right).$$

Each constraint clause identified by ℓ ($1 \leq \ell \leq h$) is a disjunction of predicates $pos/2$ and $neg/2$. The atom $pos(j, d)$ expresses that the parameter j has a value d, and the atom $neg(j, d)$ expresses its negation. The constants n_ℓ and m_ℓ ($n_\ell \geq m_\ell \geq 1$) are the number of both type of atoms and *pos* atoms, respectively, for each clause ℓ. Then *CMCA* has the following properties:

- $a_{ij} \in \{0, 1, 2, \ldots, v_j - 1\}$,
- every row r ($1 \leq r \leq N$) satisfies constraint $\mathcal{C}$ (called *domain constraints*).

$$\bigwedge_{1 \leq r \leq N, 1 \leq \ell \leq h} \left(\bigvee_{1 \leq m \leq m_\ell} (a_{r, j_m^\ell} = d_m^\ell) \quad \vee \bigvee_{m_\ell + 1 \leq n \leq n_\ell} \neg (a_{r, j_n^\ell} = d_n^\ell) \right)$$

- in every $N \times t$ sub-array, all possible t-tuples (pairs when $t = 2$) that satisfy the constraint $\mathcal{C}$ occur at least once (also called *coverage constraints*).

	Display	Email	Camera	Video	Ringtones
1	8MC	**TV**	**None**	No	No
2	BW	**None**	**None**	No	No
3	16MC	**None**	**2MP**	Yes	Yes
4	16MC	None	2MP	No	No
5	16MC	**GV**	**None**	No	No
6	16MC	**TV**	**1MP**	Yes	Yes
7	BW	TV	1MP	No	No
8	8MC	**None**	**1MP**	Yes	No
9	8MC	**GV**	**1MP**	Yes	Yes

Constraints on valid configurations:

C_1: ¬(Email = GV) ∨ (Display = 16MC) ∨ (Display = 8MC)
C_2: ¬(Camera = 2MP) ∨ (Display = 16MC) ∨ (Display = 8MC)
C_3: ¬(Email = GV) ∨ ¬(Camera = 2MP)
C_4: ¬(Display = 8MC) ∨ ¬(Camera = 2MP)
C_5: ¬(Video = Yes) ∨ (Camera = 2MP) ∨ (Camera = 1MP)
C_6: ¬(Video = Yes) ∨ (Display = 16MC) ∨ (Display = 8MC)
C_7: ¬(Ringtones = Yes) ∨ ¬(Video = No)
C_8: ¬(Display = 16MC) ∨ ¬(Email = TV) ∨ ¬(Camera = 2MP)

Fig. 2. An optimal test suite of strength-2 CCT

A *CMCA* of size N is *optimal* if N is equal to the smallest n such that a *CMCA* of size n exists. We refer to a t-tuple as a *valid* t-tuple if it satisfies the constraint $\mathcal{C}$, otherwise we call it *invalid*. Note that the constraint $\mathcal{C}$ is defined as hard constraint, and we have no soft constraints in this definition.

As an illustration, we use a simplified software product line of mobile phones proposed in [8]. The product line of Fig. 1 has five configuration options, three of which can have three values, while others have two choices of values (Yes and No). "Display" has exactly one value among 16MC, 8MC, and BW. The product line has seven constraints on valid configurations. The constraint (3) forbids a pair $(\mathrm{GV}, 2\mathrm{MP})$ in the interaction of $(\mathrm{Email}, \mathrm{Camera})$. Exhaustive testing requires $2^2 \times 3^3 = 108$ test cases (or configurations) for all different phones produced by instantiating this product line. The constraints reduce the number of test cases to 31. However, in general, such testing fails to scale to large product lines. Instead, strength-t CCT is able to provide effective testing while avoiding the combinatorial explosion. The question is what is the smallest number of test cases for the product phone line. An optimal test suite of strength-2 CCT is shown in Fig. 2. It consists of 9 test cases, and gives an answer to the question. Each row represents an individual test case, which satisfies the domain constraints in (1–7). Each column represents a configuration option. In the interaction of (Email, Camera), we highlight the different pairs to show that all valid pairs (7 combinations) occur at least once. Note that invalid pairs $(\mathrm{GV}, 2\mathrm{MP})$ and $(\mathrm{TV}, 2\mathrm{MP})$ do not occur. This property holds for all interactions of two configuration options

(Display, Email)	(Display, Camera)	(Display, Video)	(Display, Ringtones)	(Email, Camera)
(16MC, GV)	(16MC, 2MP)	(16MC, Yes)	(16MC, Yes)	(GV, 2MP) ♠
(16MC, TV)	(16MC, 1MP)	(16MC, No)	(16MC, No)	(GV, 1MP)
(16MC, None)	(16MC, None)	(8MC, Yes)	(8MC, Yes)	(GV, None)
(8MC, GV)	(8MC, 2MP) ♠	(8MC, No)	(8MC, No)	(TV, 2MP) ♠
(8MC, TV)	(8MC, 1MP)	(BW, Yes) ♠	(BW, Yes) ♠	(TV, 1MP)
(8MC, None)	(8MC, None)	(BW, No)	(BW, No)	(TV, None)
(BW, GV) ♠	(BW, 2MP) ♠			(None, 2MP)
(BW, TV)	(BW, 1MP)			(None, 1MP)
(BW, None)	(BW, None)			(None, None)
(Email, Video)	(Email, Ringtones)	(Camera, Video)	(Camera, Ringtones)	(Video, Ringtones)
(GV, Yes)	(GV, Yes)	(2MP, Yes)	(2MP, Yes)	(Yes, Yes)
(GV, No)	(GV, No)	(2MP, No)	(2MP, No)	(Yes, No)
(TV, Yes)	(TV, Yes)	(1MP, Yes)	(1MP, Yes)	(No, Yes) ♠
(TV, No)	(TV, No)	(1MP, No)	(1MP, No)	(No, No)
(None, Yes)	(None, Yes)	(None, Yes) ♠	(None, Yes) ♠	
(None, No)	(None, No)	(None, No)	(None, No)	

Fig. 3. All pairs of configuration options

and thus satisfies the coverage constraints. This test suite is an instance of an optimal $CMCA(2, 5, 3^3 2^2, \mathcal{C})$ of size 9, where the notation $3^3 2^2$ is an abbreviation of $(3, 3, 3, 2, 2)$. The CNF formula $\mathcal{C}$ consists of eight constraint clauses shown at the bottom of Fig. 2. The notation $(j = d)$ and $\neg(j = d)$ is used for convenience, instead of $pos(j, d)$ and $neg(j, d)$. The clause C_ℓ represents the constraint (ℓ) for $1 \leq \ell \leq 4$ in Fig. 1. The conjunction of C_5 and C_6 represents constraint (5). The clauses C_7 and C_8 represent the constraints (6) and (7) respectively. As can be seen in Fig. 3, this array has 67 pairs in total for all interactions of parameter values, 57 of which are valid, while other 10 (followed by ♠) are invalid. For example, (GV, 2MP) in the interaction of (Email, Camera) is an invalid pair which is directly derived from C_3. (BW, Yes) in the interaction of (Display, Ringtones) is an implicit invalid pair which is derived from C_6, C_7, and the constraint that each parameter must have exactly one value. Such implicit invalid pairs make it difficult to find all valid (or invalid) pairs manually. Thus, current existing implementations calculate them at preprocessing.

3 The *catnap* Approach

We begin with describing *catnap*'s fact format for *CMCA* instances and then present ASP encodings for *CMCA* finding. Due to lack of space, *catnap* encodings presented here are restricted to strength $t = 2$ and stripped off capacities for handling $t \geq 3$. We also omit the explanation of calculating valid tuples at prepossessing.

Fact Format. Facts express the parameter values and constraints of a *CMCA* instance in the syntax of ASP grounders, in our case *gringo*. Their format can be easily explained via the phone product line in Section 2. Its fact representation is shown in Listing 1. The facts of the predicate `p/2` provide pa-

```
1  p("Display",("16MC"; "8MC"; "BW")). p("Email",("GV"; "TV"; "None")).
2  p("Camera",("2MP"; "1MP"; "None")). p("Video",("Yes"; "No")). p("Ringtones",("Yes"; "No")).

4  c(1,(neg("Email","GV"); pos("Display","16MC"); pos("Display","8MC"))).
5  c(2,(neg("Camera","2MP"); pos("Display","16MC"); pos("Display","8MC"))).
6  c(3,(neg("Email","GV"); neg("Camera","2MP"))).
7  c(4,(neg("Display","8MC"); neg("Camera","2MP"))).
8  c(5,(neg("Video","Yes"); pos("Camera","2MP"); pos("Camera","1MP"))).
9  c(6,(neg("Video","Yes"); pos("Display","16MC"); pos("Display","8MC"))).
10 c(7,(neg("Ringtones","Yes"); neg("Video","No"))).
11 c(8,(neg("Display","16MC"); neg("Email","TV"); neg("Camera","2MP"))).
```

Listing 1. Facts representing the phone product line of Fig. 1

```
1  row(1..n). col(I) :- p(I,_). c(ID) :- c(ID,_).

3  1 { assigned(R,I,A) : p(I,A) } 1 :- row(R); col(I).

5  % domain constraints
6  :- not assigned(R,I,A) : c(ID,pos(I,A)); assigned(R,J,B) : c(ID,neg(J,B)); c(ID); row(R).

8  % coverage constraints
9  covered(I,J,A,B) :- assigned(R,I,A); assigned(R,J,B); I<J.
10 :- not covered(I,J,A,B); pair(I,J,A,B).
```

Listing 2. ASP encoding for strength-2 *CMCA* finding

rameter values in Line 1–2. The fact `p(`j`,`d`)` expresses that a parameter j can have a value d. Note that the ';' in the second argument is syntactic sugar, and the first fact in Line 1 is expanded into three facts `p("Display","16MC")`, `p("Display","8MC")`, and `p("Display","BW")`. The facts of the predicate `c`/2 provide constraints in Line 4–11. The fact `c(`ℓ`,`lit`)` expresses that a constraint clause identified by ℓ has literal lit. Again, the fact in Line 4 is expanded into three facts `c(1,neg("Email","GV"))`, `c(1,pos("Display","16MC"))`, and `c(1,pos("Display","8MC"))`.

First-Order Encoding. We introduce the predicate `assigned`/3 to provide the assignments of parameter values. Note that a solution is composed of a set of these assignments. The atom `assigned(`R`,`I`,`A`)` expresses that a value A is assigned to the (R, I)-entry of the array. We also use the predicate `pair`/4 to provide pre-calculated valid pairs. The atom `pair(`I`,`J`,`A`,`B`)` expresses a valid pair (A, B) in the interaction of parameters (I, J).

Our encoding for strength-2 *CMCA* finding is shown in Listing 2. Given an instance of fact format with size `n`, the rules in Line 1 generate `row(R)`, `col(I)`, and `c(ID)` for each row `R`, column `I`, and constraint `ID`. The rule in Line 3, for every row `R` and column `I`, generates a candidate of assignments at first and then constrains that there is exactly one value `A` such that `assigned(R,I,A)` holds.

For domain constraints, the rule in Line 6, for every row `R` and constraint `ID`, ensures that a value `A` is assigned to the (`R`,`I`)-entry if `c(ID,pos(I,A))` holds and value `B` is not if `c(ID,neg(J,B))` holds. As an example, for the first row and the constraint "`c(3,(neg("Email","GV");neg("Camera","2MP")))`", this rule is grounded to "`:- assigned(1,"Email","GV"),assigned(1,"Camera","2MP").`"

```
1  row(1..n). col(I) :- p(I,_). c(ID) :- c(ID,_).

3  % activation atoms
4  { activated(R) : row(R) }.
5  #minimize{ 1,R : activated(R) }.
6  :- not activated(R); activated(R+1); R>0. % can be omitted

8  1 { assigned(R,I,A) : p(I,A) } 1 :- activated(R); col(I).

10 % domain constraints
11 :- not assigned(R,I,A) : c(ID,pos(I,A)); assigned(R,J,B) : c(ID,neg(J,B)); c(ID); row(R).

13 % coverage constraints
14 covered(I,J,A,B) :- assigned(R,I,A); assigned(R,J,B); I<J.
15 :- not covered(I,J,A,B); pair(I,J,A,B).
```

Listing 3. ASP encoding for optimal strength-2 *CMCA* finding

For coverage constraints, the rule in Line 9, for every row `R`, different columns `I` and `J` (`I` $\leq$ `J`), and values `A` and `B`, generates an atom `covered(I,J,A,B)` if `assigned(R,I,A)` and `assigned(R,J,B)` hold. The atom `covered(I,J,A,B)` expresses that a pair (`A`,`B`) is covered in the interaction of parameter (`I`,`J`). Then, the rule in Line 10 ensures that a pair (`A`,`B`) in (`I`,`J`) is covered if `pair(I,J,A,B)` holds for every different columns `I` and `J`, and values `A` and `B`.

For optimal *CMCA* finding, we use the idea of *blocking variables*, in our case the predicate `activated`/1. The atom `activated(`R`)` expresses that a row R is used in a resulting array. Listing 3 shows our encoding of optimal strength-2 *CMCA* finding for a given instance with initial bound `n`. The differences from Listing 2 are Line 4–6 and 8. The rule in Line 4 generates an atom `activated(R)` for each row `R`. An optimal array can be found by minimizing the number of these atoms in Line 5. The rule in Line 8 is adjusted to generate the assignments of parameter values only for activated rows. Although the rule in Line 6 can be omitted, we keep it as an additional rule for performance improvement. We refer to the encoding of Listing 3 as *basic encoding*.

4 Extensions

We next extend the basic *catnap* encoding in view of enhancing the flexibility of multi-criteria optimization.

Easy extension for CCT under limiting resources. The basic encoding can concisely implement CCT solving based on *CMCA* as defined in Section 2. However, it does not provide CCT solving under a limited number of test cases, which happens in the real world. More precisely, the basic encoding cannot generate any test suite if the initial bound `n` is less than the minimal size. This is because the coverage constraints are not satisfied. To solve this practical issue, we weaken the coverage constraints by switching them from hard to soft (and refer to them as *weak coverage constraints*).

Listing 4 shows our encoding of strength-2 CCT solving with weak coverage constraints. The main difference from the basic encoding is that the number of covered pairs is maximized in Line 15. This encoding has two criteria,

```
1  row(1..n). col(I) :- p(I,_). c(ID) :- c(ID,_).

3  % activation atoms
4  { activated(R) : row(R) }.
5  #minimize{ 1@size,R : activated(R) }.
6  :- not activated(R); activated(R+1); R>0. % can be omitted

8  1 { assigned(R,I,A) : p(I,A) } 1 :- activated(R); col(I).

10 % domain constraints
11 :- not assigned(R,I,A) : c(ID,pos(I,A)); assigned(R,J,B) : c(ID,neg(J,B)); c(ID); row(R).

13 % coverage constraints (soft)
14 covered(I,J,A,B) :- assigned(R,I,A); assigned(R,J,B); I<J.
15 #maximize{ 1@coverage,I,J,A,B : covered(I,J,A,B) }.
```

Listing 4. ASP encoding for strength-2 CCT solving with weak coverage constraints

the minimality of size and the maximality of coverage. Their priority levels are defined by integer constants `size` and `coverage` on the right-hand side of `@` (`size`<`coverage` by default). An optimal solution can be found by a well-known multi-criteria optimization strategy called lexicographic optimization in *clingo*. It enables us to optimize criteria in a lexicographic order based on their priorities.

We refer to the encoding of Listing 4 as *weakened encoding*. The idea of weak coverage constraints allows for flexible CCT solving. The weakened encoding can generate test suites of maximal coverage under limiting resources if an initial bound `n` is less than the minimal size, otherwise it can find optimal *CMCA*s. Moreover, it does not require the pre-calculation of valid pairs which existing implementations rely upon. From a viewpoint of hybridization, the weakened encoding proposes a complementary approach to *prioritized CT* [6, 18], which is an extension of CT with ordering (or re-ordering) strategies between test cases for detecting failures as early as possible.

Easy extension of CCT with soft constraints. Soft constraints are useful to express preferences and costs in a wide range of combinatorial optimization problems. However, there are few implementations that provide CCT solving with soft constraints. We here extend *catnap*'s fact format and weakened encoding with soft constraints. For this, we utilize the idea of *constraint atoms* used for timetabling [2]. The basic encoding can be extended in the same way.

Constraint atoms are instances of two predicates: `hard_constraint`/1 and `soft_constraint`/2. The atom `hard_constraint(`C`)` expresses that a constraint clause C is a hard constraint. The atom `soft_constraint(`C`,`W`)` expresses that C is a soft constraint and its weight is W. Listing 5 shows an extension of the fact representation of the phone product line with constraint atoms. In this case, new constraint clauses in Line 3–4 are added as soft constraints.

Extending the weakened encoding with constraint atoms can be done by replacing the rule in Line 11 of Listing 4 with the rules shown in Listing 6. For hard domain constraints, the rule in Line 2–3 is the same as before except `hard_constraint(ID)`. For soft domain constraints, the rule in Line 6–8 generates an atom `penalty(ID,R,W)` if the assignments in `R` violate a constraint `ID` for every row `R` and soft constraint identified by `ID` of weight `W`. That is, for

```
1 hard_constraint(1..8).
2 soft_constraint(9..10,1).
3 c(9,(neg("Display","BW"); pos("Camera","None"))).
4 c(10,(neg("Camera","None"); neg("Display","BW"); neg("Email","None"))).
```

Listing 5. An extended fact representation with constraint atoms

```
1 % domain constraints (hard)
2 :- not assigned(R,I,A) : c(ID,pos(I,A)); assigned(R,J,B) : c(ID,neg(J,B));
3    hard_constraint(ID); row(R).
5 % domain constraints (soft)
6 penalty(ID,R,W) :- not assigned(R,I,A) : c(ID,pos(I,A));
7                       assigned(R,J,B) : c(ID,neg(J,B));
8                    soft_constraint(ID,W); row(R).
9 #minimize{ W@soft,ID,R : penalty(ID,R,W) }.
```

Listing 6. An extended domain constraints with constraint atoms

each violation in `R` for `ID`, the atom `penalty(ID,R,W)` is generated. Then, the number of these atoms is minimized in Line 9, where its priority level is defined by an integer constant `soft`.

The resulting encoding has three criteria, the minimality of size, the maximality of coverage, and the minimality of penalty costs. The default ordering of priority levels is `soft`<`size`<`coverage`, but can be changed by the command line option of *clingo*. As an example, in a case of `soft`=`size`, the encoding can generate optimal test suites of minimal sum of size and penalty costs.

CCT solving with *catnap* can be promising, since it allows for flexible CCT solving of generating optimal test suites by varying a set of hard and soft constraints, switching them between hard and soft, and varying the priority levels of criteria, even if the number of test cases is limited.

5 Experiments

As we have mentioned, *catnap* accepts a CCT instance in fact format and combines it with a first-order ASP encoding for CCT solving, which is subsequently solved by the ASP system *clingo* that returns an assignment representing a solution to the original CCT instance. The *catnap* system also accepts the *CASA* format [12]. For this, we implemented a converter that provides us with the resulting CCT instance in *catnap*'s fact format.

For our experiments, we use all 35 instances[5] proposed in [8], five of which are from highly configurable software systems such as *SPIN*, *GCC*, *Apache*, and *Bugzilla*. Note that these are *CMCA* instances in *CASA* format which have no soft constraints. We ran them on a multi-core Linux machine equipped with Xeon 3.16GHz and 32GB RAM. We imposed a time-limit ($t.o$) of 1 hour for each run.

[5] `http://cse.unl.edu/~citportal/public/tools/casa/benchmarks.zip`

Table 1. Comparison between the basic and weakened encodings

instance	t	k	$(v_1,\ldots,v_k)$	$\lvert\mathcal{C}\rvert$	weakened	basic
benchmark_apache	2	172	$2^{158}3^{8}4^{4}5^{1}6^{1}$	7	**30**	**30**
benchmark_bugzilla	2	52	$2^{49}3^{1}4^{2}$	5	**16**	**16**
benchmark_gcc	2	199	$2^{189}3^{10}$	40	**15**	**15**
benchmark_spins	2	18	$2^{13}4^{5}$	13	**19**	**19**
benchmark_spinv	2	55	$2^{42}3^{2}4^{11}$	49	**31**	**31**
benchmark_1	2	97	$2^{86}3^{3}4^{1}5^{5}6^{2}$	24	**38**	**38**
benchmark_2	2	94	$2^{86}3^{3}4^{3}5^{1}6^{1}$	22	**30**	**30**
benchmark_3	2	29	$2^{27}4^{2}$	10	**18**	**18**
benchmark_4	2	58	$2^{51}3^{4}4^{2}5^{1}$	17	**20**	*t.o*
benchmark_5	2	174	$2^{155}3^{7}4^{3}5^{5}6^{4}$	39	**46**	**46**
benchmark_6	2	77	$2^{73}4^{3}6^{1}$	30	**24**	*t.o*
benchmark_7	2	30	$2^{29}3^{1}$	15	**9**	**9**
benchmark_8	2	119	$2^{109}3^{2}4^{2}5^{3}6^{3}$	37	**37**	38
benchmark_9	2	61	$2^{57}3^{1}4^{1}5^{1}6^{1}$	37	**20**	*t.o*
benchmark_10	2	147	$2^{130}3^{6}4^{5}5^{2}6^{4}$	47	**41**	**41**
benchmark_11	2	96	$2^{84}3^{4}4^{2}5^{2}6^{4}$	32	**40**	41
benchmark_12	2	147	$2^{136}3^{4}4^{3}5^{1}6^{3}$	27	**36**	**36**
benchmark_13	2	133	$2^{124}3^{4}4^{1}5^{2}6^{2}$	26	**36**	**36**
benchmark_14	2	92	$2^{81}3^{5}4^{3}6^{3}$	15	**36**	**36**
benchmark_15	2	58	$2^{50}3^{4}4^{1}5^{2}6^{1}$	22	**30**	**30**
benchmark_16	2	87	$2^{81}3^{3}4^{2}6^{1}$	34	**24**	*t.o*
benchmark_17	2	137	$2^{128}3^{3}4^{2}5^{1}6^{3}$	29	**36**	**36**
benchmark_18	2	141	$2^{127}3^{2}4^{4}5^{6}6^{2}$	28	**41**	**41**
benchmark_19	2	197	$2^{172}3^{9}4^{9}5^{3}6^{4}$	43	**44**	*t.o*
benchmark_20	2	158	$2^{138}3^{4}4^{5}5^{4}6^{7}$	48	59	**54**
benchmark_21	2	85	$2^{76}3^{3}4^{2}5^{1}6^{3}$	46	**36**	**36**
benchmark_22	2	79	$2^{72}3^{4}4^{1}6^{2}$	22	**36**	**36**
benchmark_23	2	27	$2^{25}3^{1}6^{1}$	15	**12***	**12***
benchmark_24	2	119	$2^{110}3^{2}5^{3}6^{4}$	29	43	**42**
benchmark_25	2	134	$2^{118}3^{6}4^{2}5^{2}6^{6}$	27	**48**	50
benchmark_26	2	95	$2^{87}3^{1}4^{3}5^{4}$	32	**27**	**27**
benchmark_27	2	62	$2^{55}3^{2}4^{2}5^{1}6^{2}$	20	**36**	**36**
benchmark_28	2	194	$2^{167}3^{16}4^{2}5^{3}6^{6}$	37	**50**	51
benchmark_29	2	144	$2^{134}3^{7}5^{3}$	22	**25**	**25**
benchmark_30	2	79	$2^{73}3^{3}4^{3}$	35	**16**	**16**
#best					33	26

We used the ASP system *clingo* (version 4.5.3) with the multi-threaded portfolio search of four configurations.[6]

At first, we analyze the difference between the basic and weakened encodings. Table 1 contrasts the bounds obtained from both encodings. The information of the $CMCA(t,k,(v_1,\ldots,v_k),\mathcal{C})$ instances is given in the first five columns. We highlight the best bound of different encodings for each instance. The #best row gives the number of best bounds for each encoding. The symbol '*' indicates that *catnap* proved the optimality of the obtained bound. The weakened encoding was able to find the best bounds for 33 instances compared with 26 obtained with the basic encoding. The basic encoding found no solution to 5 instances in the time limit. Both encodings were able to prove that the previous known bound (12) for `benchmark_23` is optimal. Because of these observations, we adopt the weakened encoding as the default setting of *catnap*.

[6] The combination of `--config={trendy,jumpy}` and `--opt-strat={bb,1,usc,1}`

Table 2. Comparison of *catnap* with other approaches

instance	*catnap*	*TCA*	*CASA*	*Cascade*	*ACTS*
benchmark_apache	**30**	**30**	32	*n.a*	33
benchmark_bugzilla	**16**	**16**	**16**	20	19
benchmark_gcc	**15**	16	19	*n.a*	23
benchmark_spins	**19**	**19**	**19**	27	26
benchmark_spinv	**31**	**31**	36	41	45
benchmark_1	38	**36**	38	*n.a*	48
benchmark_2	**30**	**30**	**30**	*n.a*	32
benchmark_3	**18**	**18**	**18**	19	19
benchmark_4	**20**	**20**	**20**	24	22
benchmark_5	46	**43**	45	*n.a*	54
benchmark_6	**24**	**24**	**24**	30	25
benchmark_7	**9**	**9**	**9**	12	12
benchmark_8	**37**	**37**	38	*n.a*	47
benchmark_9	**20**	**20**	**20**	23	22
benchmark_10	41	**40**	42	*n.a*	47
benchmark_11	40	**39**	41	*n.a*	47
benchmark_12	**36**	**36**	39	*n.a*	43
benchmark_13	**36**	**36**	**36**	*n.a*	40
benchmark_14	**36**	**36**	37	*n.a*	39
benchmark_15	**30**	**30**	**30**	37	32
benchmark_16	**24**	**24**	**24**	*n.a*	25
benchmark_17	**36**	**36**	38	*n.a*	41
benchmark_18	41	**39**	41	*n.a*	52
benchmark_19	44	**43**	47	*n.a*	51
benchmark_20	59	**49**	52	*n.a*	60
benchmark_21	**36**	**36**	**36**	*n.a*	39
benchmark_22	**36**	**36**	**36**	*n.a*	37
benchmark_23	**12**	**12**	**12**	14	14
benchmark_24	43	**40**	42	*n.a*	48
benchmark_25	48	**45**	47	*n.a*	52
benchmark_26	**27**	**27**	30	*n.a*	34
benchmark_27	**36**	**36**	**36**	45	37
benchmark_28	50	**47**	50	*n.a*	57
benchmark_29	**25**	**25**	29	*n.a*	29
benchmark_30	**16**	**16**	19	*n.a*	22
#best	25	34	15	0	0

Next, we compare the performance of *catnap* with other approaches. Table 2 contrasts the bounds obtained by *catnap* with the best known ones in [14] obtained from dedicated implementations: *CASA* [12], *TCA* [14], *ACTS* [21], and *Cascade* [22]. *CASA* and *TCA* are metaheuristics-based dedicated implementations. *ACTS* and *Cascade* are based on greedy algorithms. The symbol '*n.a*' indicates that a solver found no solution in [14]. The *catnap* system was able to find the best bounds for 25 instances, compared with 34 of *TCA*, 15 of *CASA*, and 0 of *Cascade* and *ACTS*. For the large instance `benchmark_gcc`, *catnap* was able to find a better bound (`15`) than the others. Although it does not fully match the performance of *TCA*, *catnap* can be competitive to *CASA* and can outperform *Cascade* and *ACTS*.

Finally, we discuss some more details of our experimental results. We used a simple configuration for multi-threaded portfolio search of *clingo*, but it took a longer time to find the best bounds of many instances than the others. This is a limitation of our approach at present. To overcome this issue, we will investigate the best configuration of *clingo*, since it offers several optimization strategies. Evaluating the scalability of *catnap* for higher strength $t \geq 3$ is also an important future work.

6 Conclusion

From an ASP perspective, the most relevant related works are ASP encodings of *event-sequence testing* [3, 5, 11]. Event-sequence testing is a testing technique especially for event-driven systems and is different from CCT that focuses on highly configurable systems. The ASP encodings in [5, 11] use `#maximize` statements including `covered` atoms for two purposes. One is to generate single test cases of maximal coverage in each iteration of ASP-based greedy algorithms. *Cascade* adopts a similar technique. Another is to generate strength-t test suites of maximal $(t+1)$-coverage for early failure detection. This technique is closely related to prioritized CT [18]. Note that the purpose of the weak coverage constraints in Section 4 is to provide CCT under limiting resources.

We presented an ASP-based approach to solving the CCT problems. The resulting system *catnap* consists of first-order encodings and delegates solving tasks to general-purpose ASP systems. We showed that ASP is an ideal modeling language for combinatorial testing, as demonstrated by *catnap*'s compact encodings for CCT solving. We contrasted the performance of *catnap* to the best known bounds obtained via dedicated implementations. The *catnap* system demonstrated that ASP's general-purpose technology allows us to compete with state-of-the-art CCT solving techniques. All source code of *catnap* is available from `https://potassco.org/doc/apps/`.

Closely related to constraint solving in CCT, recent advances in Constraint ASP (CASP; [1, 13]) open up a successful direction to extend ASP to be more expressive. CASP solvers such as *clingcon* can solve finite linear Constraint Satisfaction Problems in a declarative way. We will investigate the possibilities of CCT solving with CASP to extend CCT with richer constraints.

References

1. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In: Faber, W., Lee, J. (eds.) Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09). pp. 16–30 (2009)
2. Banbara, M., Inoue, K., Kaufmann, B., Schaub, T., Soh, T., Tamura, N., Wanko, P.: teaspoon: Solving the curriculum-based course timetabling problems with answer set programming. In: Proceedings of the Eleventh International Conference of the Practice and Theory of Automated Timetabling (PATAT'16). pp. 13–32 (2016)
3. Banbara, M., Tamura, N., Inoue, K.: 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, vol. 17, pp. 86–97. Schloss Dagstuhl (2012)
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
5. Brain, M., Erdem, E., Inoue, K., Oetsch, J., Pührer, J., Tompits, H., Yilmaz, C.: Event-sequence testing using answer-set programming. International Journal on Advances in Software 5(3-4), 237–251 (2012)

6. Bryce, R.C., Colbourn, C.J.: Prioritized interaction testing for pair-wise coverage with seeding and constraints. Information and Software Technology 48(10), 960–970 (2006)
7. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatiorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1997)
8. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering 34(5), 633–650 (2008)
9. Colbourn, C.J., Dinitz, J.H.: Handbook of Combinatorial Designs, Second Edition. Chapman & Hall/CRC (2006)
10. Czerwonka, J.: Pairwise testing in real world. practical extensions to test case generators. In: Proceedings of the 24th Annual Pacific Northwest Software Quality Conference. pp. 419–430 (2006)
11. Erdem, E., Inoue, K., Oetsch, J., Pührer, J., Tompits, H., Yilmaz, C.: 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). pp. 25–34. Xpert Publishing Services (2011)
12. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: An improved meta-heuristic search for constrained interaction testing. In: Proceedings of the first International Symposium on Search Based Software Engineering (SSBSE 2009). pp. 13–22. IEEE (2009)
13. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Hill, P., Warren, D. (eds.) Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). pp. 235–249. Springer-Verlag (2009)
14. Lin, J., Luo, C., Cai, S., Su, K., Hao, D., Zhang, L.: TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation. In: Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015). pp. 494–505. IEEE (2015)
15. Mandl, R.: Orthogonal latin squares: An application of experiment design to compiler testing. Communications of the ACM 28(10), 1054–1058 (1985)
16. Nanba, T., Tsuchiya, T., Kikuno, T.: Constructing test sets for pairwise testing: A SAT-based approach. In: Proceedings of the Second International Conference on Networking and Computing (ICNC 2011). pp. 271–274. IEEE (2011)
17. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Computing Surveys 43(2), 11:1–11:29 (2011), http://doi.acm.org/10.1145/1883612.1883618
18. Petke, J., Yoo, S., Cohen, M.B., Harman, M.: Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In: Proceedings of ESEC/FSE 2013. pp. 26–36. ACM (2013)
19. Tatsumi, K.: Test case design support system. In: Proceedings of the International Conference on Quality Control (ICQC 1987). pp. 615–620 (1987)
20. Yamada, A., Kitamura, T., Artho, C., Choi, E., Oiwa, Y., Biere, A.: Optimization of combinatorial testing by incremental SAT solving. In: Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015). pp. 1–10. IEEE (2015)
21. Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R., Kuhn, D.R.: An efficient algorithm for constraint handling in combinatorial test generation. In: Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013). pp. 242–251. IEEE (2013)
22. Zhang, Z., Yan, J., Zhao, Y., Zhang, J.: Generating combinatorial test suite using combinatorial optimization. Journal of Systems and Software 98, 191–207 (2014)