APPLICATIONS OF UNIFIED COMBINATORIAL INTERACTION TESTING

by
OĞUZ ÖZSAYGIN

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2021

APPLICATIONS OF UNIFIED COMBINATORIAL INTERACTION TESTING

Approved by:

Date of Approval: July 08, 2021

ABSTRACT

APPLICATIONS OF UNIFIED COMBINATORIAL INTERACTION TESTING

OĞUZ ÖZSAYGIN

COMPUTER SCIENCE AND ENGINEERING M.Sc. THESIS, JULY 2021

Thesis Supervisor: Assoc. Prof. Cemal Yılmaz

Keywords: combinatorial testing, covering arrays, constraint solving, answer set programming

U-CIT approach has provided a flexible and systematic method to flexibly define and compute combination interaction testing (CIT) objects generating covering arrays used in software testing. By U-CIT, software system under test and coverage criterion are expressed as a constraint solving problem, and CIT objects are computed by using appropriate constraint solvers. The convenience of defining flexibly coverage criteria brought by U-CIT has made it possible to easily define new CIT objects to test any software system. Although U-CIT objects are generated by solving constraints with constraint solvers in these studies, a higher level modelling abstraction may be required to define complex system models and coverage criteria. In this study, we present UCIT-ASP approach that we developed to generate U-CIT objects by using Answer Set Programming (ASP) which is a declarative modeling language. In addition, by using ASP modeling libraries that was developed within the scope of this study, we both generated U-CIT objects already defined in the literature (i.e. standard covering arrays, test case aware covering arrays, etc.) and defined new U-CIT objects, specifically for graph-based systems (for the testing of mobile applications, multi-threaded systems, etc.). In our case studies to experience with UCIT-ASP on the generation of well-known CIT objects, we have observed that our approach generated smaller CIT objects than specialized covering array generation methods in the literature at the cost of computation times.

ÖZET

TÜMLEŞİK KOMBİNEZON ETKİLEŞİM SINAMA YÖNTEMİNİN UYGULAMALARI

OĞUZ ÖZSAYGIN

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, TEMMUZ 2021

Tez Danışmanı: Doç. Dr. Cemal Yılmaz

Anahtar Kelimeler: kombinezon etkileşim sınama, kapsayan diziler, kısıt çözme, bildirimli modelleme

T-KES yaklaşımı, yazılımların test edilmesinde kullanılan kapsayan dizileri oluşturan kombinezon etkileşim sınama (KES) objelerini esnek şekilde tanımlamak ve hesaplamak için bir esnek ve sistematik yöntem sağladı. Bu yaklaşım sayesinde test altındaki yazılım sistemleri ve kapsama kriterleri bir kısıt çözme problemi olarak ifade edilip, hali hazırda kullanılan kısıt çözücüler kullanılarak KES objeleri hesaplanmaktadır. T-KES'in getirdiği bu esnek kapsama kriteri tanımlayabilme rahatlığı, herhangi bir yazılım sistemini test edebilmek için yeni KES objelerinin kolayca tanımlanabilmesine olanak sağlamıştır. Bu çalışmalarda her ne kadar kısıt çözücüler kullanılarak T-KES objeleri üretilse de, karmaşık sistem modellerinin ve kapsama kriterlerinin tanımlanması için daha yüksek seviyede modelleme soyutlaması yapılması gerekebilir. Bu çalışmada, T-KES objelerini bildirimli bir modelleme dili olan ASP kullarak üretmek için geliştirdiğimiz TKES-ASP yaklaşımını sunuyoruz. Ayrıca, bu çalışma kapsamında geliştirilen ASP modelleme kütüphanelerini kullanarak hem hali hazırda olan T-KES objelerini üretip hem de, özellikle çizge tabanlı sistemler için (mobil uygulamalar, multi-threaded sistemler, vs.) için T-KES objeleri tanımlıyoruz. Literatürde tanımlanan KES objelerinin TKES-ASP kullanılarak yeniden üretilmesini konu alan vaka çalışmalarında, üretilen KES objelerinin hem akademik hem de endüstride sıkça kullanılan kapsayan dizi üretme yöntemlerinden daha fazla sürede bu dizileri hesaplaması karşılığında daha küçük boyutlu objeler ürettiği gözlemlenmiştir.

ACKNOWLEDGEMENTS

This research was supported by the Scientific and Technological Research Council of Turkey (118E204).

I would first like to thank my supervisor, Assoc. Prof. Cemal Yılmaz for his endless support, knowledge and guidance.

I would like to thank to the all jury members Assoc. Prof. Hüsnü Yenigün and Assoc. Prof. Hasan Sözer for their precious time and insightful comments.

I would like to also thank to my colleague Hanefi Mercan for his technical support and helpful feedback during my study.

I would like express my special thanks to my family and friends supported me throughout this process.

To my family & friends

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Nowadays, software systems have many parameters to be tested such as the interaction of user inputs, configuration parameters, and multi-threaded inter-leavings. While these parameters provide flexibility to end-users, they cause serious problems on ensuring the quality assurance of software. The reason is that it is a challenging problem to test all configuration parameter space (if not possible at all), even on a small scale. For example, Apache server has 172 configuration parameters and it has $1.8x10^{55}$ configuration combinations which can be obtained from the combinations of these parameters. Even if each configuration is tested for 1 second, testing all possible configurations might take longer than millions of years. Hence, it is not feasible to test all possible configurations for the Apache server or similar software systems.

For similar reasons, the testing of industrial software systems is almost always performed using a subset selected by a sampling method from a very large configuration space (combination of parameter values). This sample is considered as having the ability to represent the entire configuration space. Practically, this sampling functionality is usually performed by using techniques gathered under the name of Combination Interaction Testing (CIT) methods (Yilmaz (2013); Nie & Leung (2011)). CIT methods have two main inputs: configuration space and coverage criterion. The configuration space contains all valid configurations. The coverage criterion identifies all valid combinations of parameter values to be tested. A CIT object is computed as an output for a given configuration space and coverage criterion. This object is a set of configurations selected from the configuration space in a way that it obtains full coverage under the coverage criterion. Once a CIT object is computed, the testing of the software system in interest can be achieved by testing the selected set of configurations.

Many studies in the literature show that CIT methods are used successfully in many different application areas (Williams & Probert (1996); Schroeder, Faherty & Korel (2002); Yilmaz, Cohen & Porter (2004); Johansen, Haugen & Fleurey (2012); Lei, Carver, Kacker & Kung (2007); Yuan, Cohen & Memon (2011)). However, CIT

methods are still used far below their potential. The reason is that the existing CIT objects only support very limited test scenarios. Moreover, it is very difficult, if not impossible, to use these objects in cases where test scenarios at hand differ slightly from supported scenarios. Therefore, different CIT problems require the development of specialized CIT constructors to compute relevant CIT objects. However, by considering the fact that there are at least 50 articles/papers in the literature used to compute covering arrays, it can be understood how difficult and a costly process to develop specialized CIT objects and methods to compute these objects (Yilmaz, Fouche, Cohen, Porter, Demiroz & Koc (2014); Nie & Leung (2011)).

To bring flexibility to the definition process of new CIT objects, Mercan, Javeed & Yilmaz (2020) has introduced Unified Combinatorial Interaction Testing (U-CIT) approach which allows practitioners to define their own CIT objects to test a specific configuration system under any specific coverage criterion. In U-CIT approach, both the system under test and the coverage criterion are expressed as constraints. While already existing CIT constructor uses constraints to define invalid parameters values or parameter value combinations, in U-CIT, constraints are used to express both entities to be covered and test cases. Specifically, an entity to be tested to achieve full coverage under a given coverage criterion is referred as a testable entity.

For example, consider t-way standard covering arrays for a given configuration space model having parameters that take a value from its discrete domain. While computing a t-way covering array, each t-tuple combination to be covered is marked as a testable entity in U-CIT. While computing a CIT object, the set of testable entities are divided into subsets in a way that when entities in each subset are solved with system model constraints via a constraint solver (a SAT solver or CSP solver), and the satisfiable solution obtained at the result of this process is referred as a test case. Each satisfiable solution specific to the entity subset composes a U-CIT object.

However, constraints to be provided to solvers must be in a form similar to Boolean expressions. Although a front-end is provided by constraint solvers to help to model system models and coverage criteria with abstractions up to a certain level, while modelling more complex coverage criteria and software systems, expressing constraints with Boolean expressions is a burden for practitioners. Also, each solvers recognize their own syntax. However, this creates an obstacle to practitioners to switch between solvers while generating U-CIT objects.

In this paper, we introduce UCIT-ASP approach to model software systems and coverage criteria declaratively by using Answer Set Programming (ASP) as a front-end. The motivation beneath introducing UCIT-ASP approach is to provide a front-end system to practitioners to model their own coverage criterion with high-level

abstraction. Moreover, we developed several modeling libraries allowing to test a software system for well-known CIT scenarios by modelling these CIT objects as U-CIT objects. In addition to these objects, we introduce two new U-CIT objects to test graph-based software systems, which are path aware covering arrays and def-use pair covering arrays.

The rest of the paper is organized as follows: Section 2 provides background information about technologies and concepts used in this work, Section 3 presents our contributions by going through our proposed approach, Section 4 presents the experiments evaluating the proposed approach on different studies, Section 5 presents the possible threats to the validity of approach proposed, Section 6 discusses related work and finally Section 7 presents concluding remarks and discusses possible future work.

2. BACKGROUND INFORMATION

2.1 Combinatorial Interaction Testing

Combinatorial Interaction Testing approaches (CIT) usually describe a software system under test as a set of parameters that takes a value from its value set. In a real-life scenario, certain aspects of software systems cannot be tested for certain parameter values or parameter value combinations. Any software system can contain such constraints which invalidate certain parameter combinations. For a given such software system, CIT approach by taking system parameters and invalid parameter combinations as an input, it generates a set of test cases, which also referred as CIT object, where test cases meet requirements provided under a given coverage criterion. These test cases are usually composed of the combinatorial interaction of parameter values of the given configuration space model.

For instance, t-way covering arrays, which is one of a well-known CIT object in the literature, covers all valid t-tuples of given configuration space model at least once where t is the coverage strength for parameter interactions. The motivation of using this object is to reveal all the failures occurring due to the interaction between t or fewer parameters.

2.2 Unified Combinatorial Interaction Testing

CIT objects defined in the literature are not always adequate to test specific requirements for different software systems. Therefore, recently many new CIT objects have

been introduced to test systems by considering different CIT problems (Demiroz & Yilmaz (2012); Yilmaz (2013)).

U-CIT allows defining our CIT objects under any coverage criterion for any type of software system. Fundamentally, it defines a software system and entities generated under a coverage criterion as constraints by representing them as a *cov-CSP* problem which is a constraint solving problem to obtain a full-coverage under given coverage criterion.

A satisfiable solution of system constraints and a subset of entities produced by a constraint solver is referred as a test case. Moreover, all solutions obtained by solving the entity subsets and the system model composes a U-CIT object (Mercan et al. (2020)).

As aforementioned, system model constraints and entity constraints are two main constraints. System constraints involve the characteristic attributes of the system under test by considering the specifications of a coverage criterion. In the definition of the system model as U-CIT constraints, all system attributes required for the given coverage criterion must be defined as system constraints. In other words, software systems must be formulated as constraints in a way that testable entities can be expressed with respect to these constraints. A testable entity definition is also provided in form of constraints which states what configurations must be covered to increase the coverage. A testable entity represents a single entity to be tested under a given coverage criterion (Mercan et al. (2020)). For example, in the computation of a standard t-way covering array, a testable entity is expressed as a single t-tuple that involves parameter value combinations of t configuration parameters. In this example, all t-tuple combinations to be covered compose the overall list of testable entities.

In U-CIT, a test case is a solution satisfying the system model and a subset of testable entity, when constraints of them are solved by a constraint solver. Simply, U-CIT invokes a constraint solver (specifically an SAT solver or CSP solver) to find a satisfiable solution to these constraints. The solution generated by the solver for each subset is considered as a valid test case under the given coverage criterion (Mercan et al. (2020)).

U-CIT approach uses two covering array construction methods (covering array constructors) to generate U-CIT objects: *generate-and-cover* and *cover-and-generate*. The basic logic of *generate-and-cover* method is to generate a new test case covering entities yet to be covered by ensuring if each testable entity is covered by test cases that have already been generated. On the other hand, *cover-and-generate* method

aims to generate a cluster of testable entities by trying to collect as many as testable entities in the same cluster that can be satisfied together (in other words, covered in the same test case). Each entity cluster that is solved by a constraint solver corresponds to a test case (Mercan et al. (2020)).

In our studies, instead of using the constructors aforementioned, we have introduced *one-test-case-at-a-time* constructor and generated U-CIT objects with it. In comparison with other U-CIT constructors, given a time limit, *one-test-case-at-a-time* constructor generates test cases with an approach that tries to cover the most number of testable entities selected from the entity list in each iteration and continues to iterate until no entity is left to be covered.

In the U-CIT paper, many new CIT objects have been already defined with U-CIT approach such as order-based, structural, and usage-based CIT objects (Mercan et al. (2020)). In the rest of this paper, we will also introduce new CIT objects by using U-CIT approach, such as def-use pair covering arrays and path aware covering arrays.

2.3 Answer Set Programming

Answer Set Programming (ASP) is a declarative programming language that allows defining difficult search problems as a constraint satisfaction problem. Basically, in ASP a search problem is expressed as a set of ASP constraints referred as *rules*. In the definition of a problem statement, the problem is expressed as a logic problem where a set of values and rules satisfying all rules defined in the problem statement is referred as a solution.

In ASP, a satisfiable solution is generated by an ASP solver. Since the search problem in constrained search space is a generic problem, many ASP solvers have been already defined in the literature such as clasp (Gebser, Kaufmann & Schaub (2012)), assat (Lin & Zhao (2004)).

In ASP, each ASP rule is a constraint that restraints the search space for the solver. The main purpose of these rules to narrow down possible solutions for the solver. Each rule is composed of head and body, and ends with a dot (`.`) symbol.

```
<head> :- <body>.
```

`:-` operator in the ASP expression stands for "if" statement where it creates a logical dependency between head and body parts. Specifically, this operator means that to satisfy the head of rule (as well as the rule itself), the body of the rule must be satisfied.

A rule defined without body part is referred as a *fact*.

```
<head>.
```

Another ASP syntax is choices where choices allow to solver only to pick among provided set of atoms. For example, in example rule given below, the solver can pick `a`, `b`, `c` and `d` atoms to include them in the solution.

```
{a, b, c, d}.
```

The number of choices made by the solver can be also constrained. For example, in the ASP statement below, the solver picks at least 1, at most 3 atoms from the set of atoms provided in the rule.

```
1{a, b, c, d}3.
```

Negation logic can be implemented with `:-` operator. For instance, the same choice expression can be also expressed as illustrated in the example ASP rule below. This rule states that more than 3 atoms cannot be included in the problem solution.

```
:- 3{a, b, c, d}.
```

Also, choices can be represented as values in a range. For example, following the ASP rule involving 5 atoms as choices

```
{1, 2, 3, 4, 5}.
```

which can be also expressed as illustrated in the ASP rule below:

```
{1..5}.
```

Finally, capital letters are used to represent variables in ASP. While finding a solution for a given problem with a set of rules, the ASP solver replaces variable with discrete values defined in the problem statement to obtain a satisfiable solution with problem definition. For instance, for the given problem definition below,

```
num(1). num(2). num(3).
res(X) :- res(X), X!=3.
```

an ASP solver might generate following solution:

```
res(1). res(2).
```

3. APPROACH

In this part, we will introduce UCIT-ASP approach by expressing processes involved in a U-CIT object computation using this approach. Firstly, we will express how to define coverage criterion and valid test spaces required to compute a U-CIT object by this approach. Afterwards, we will explain in detail, how a U-CIT object is computed for a given system model and coverage criterion. Finally, we will explain the development process of UCIT-ASP by going over the modelling libraries developed within the scope of this study.

U-CIT (Mercan et al. (2020)) proposes a generic approach to systematically generate CIT objects under any coverage criterion regardless of the topology of the system under test. However, in the proposed approach, the system and coverage criterion definitions are expressed with Boolean primitives. We know the fact that the front-end of a testing framework is as important as the back-end. Based on this idea, we have developed a unique approach using ASP (Gebser, Kaminski, Kaufmann & Schaub (2012)), which is the one of declarative modeling languages, so that end-users can flexibly define any coverage criterion and compute related U-CIT objects. This approach is referred as *UCIT-ASP* in the rest of the document.

In general terms, UCIT-ASP is a unique system that allows end users to define both coverage criterion and valid test case spaces in an informed and flexible way, to determine the requirements to be covered under the defined coverage criteria, and to compute U-CIT objects to cover these requirements.

3.1 Definition of Coverage Criterion

Since UCIT-ASP approach uses ASP as a front-end, it allows the use of all definition and computation capabilities of ASP language to define a coverage criterion without

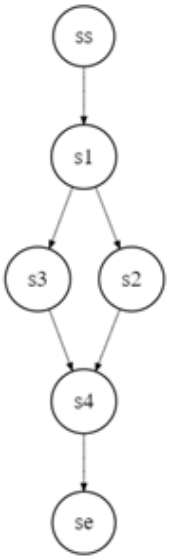


Figure 3.1 An example to graph-based models

any restrictions. In this section, we explain the UCIT-ASP method on an example without giving detailed information about ASP language. For detailed information about ASP, Section 2.3 and the literature can be referred (Gebser et al. (2012)).

```
% Start and end nodes
start_state(ss).
end_state(se).

% Nodes
state(ss).
state(s1).
state(s2).
state(s3).
state(s4).
state(se).

% Edges
edge(ss, s1).
edge(s1, s3).
edge(s3, s4).
edge(s1, s2).
edge(s2, s4).
edge(s4, se).
```

Figure 3.2 The ASP modelling of Figure 3.1

As an example, Figure 3.2 models the graph given in Figure 3.1 by using ASP language. For a given graph, nodes are defined by `state(...)` rules and edges are defined by `edge(...)` rules. The entry and exit nodes in graphs are defined by using the `start_state(...)` and `end_state(...)` rules, respectively.

For example, let's want to express graph theoretical concepts over this model. In the rest of the document, we will also use these concepts while defining novel coverage criteria on graph-based models (Section 4.4 and 4.5). Note that in this context, each path from the entry node to the exit node corresponds to a test case. The loops in the graphs are unrolled by opening these loops in the numbers requested.

```
% A start state is visited by default
visited(S) :- start_state(S)

% An edge is either taken or not taken
{taken(A, B)} :- A != B, edge(A, B), visited(A).

% At most one incoming edge to a state can be taken
:- taken(A, B), taken(C, B), A != C.

% At most one edge originating from a state can be taken
:- taken(A, B), taken(A, C), B != C.

% A state is visited iff one of the incoming edges is taken
visited(A) :- taken(_, A).

% Expressing reachability
reaches(A, B):- A != B, taken(A, B).
reaches(A, B):- A != B, taken(A, C), reaches(C, B).

% All 2-orders
any_order(A, B) :- reaches(A, B).
```

Figure 3.3 The modelling of graph-based concepts in ASP

For this scenario, the following ASP definitions can be stated as illustrated in Figure 3.3. In these definitions, the rule below always marks the entry node as a visited node.

```
visited(S) :- start_state(S).
```

The following rule is used to express the fact that each edge in the graph will either be taken or not taken on the path to be computed:

```
{taken(A, B)} :- A != B, edge(A, B), visited(A).
```

Since graphs are acyclic, at most one incoming edge to a node

```
:- taken(A, B), taken(C, B), A != C.
```

and at most one outgoing edge from a node can be taken:

```
:- taken(A, B), taken(A, C), B != C.
```

If any of the incoming edges to a node is taken, that node is considered as visited:

```
visited(A) :- taken(_, A).
```

When any of following conditions are met, node B will be reachable from node A:

```
reaches(A, B):- A != B, taken(A, B).
reaches(A, B):- A != B, taken(A, C), reaches(C, B).
```

Therefore, the following rule can be used to find all possible 2-way any node orderings:

```
any_order(A, B) :- reaches(A, B).
```

Note that each A and B node pair obtained by this rule represents valid 2-way node ordering. In addition, the coverage strength of this ordering criterion can be easily increased or similar ordering criteria can be implemented.

UCIT-ASP approach defines a special rule under the name of `entity(...)` which takes as many parameters as needed for the end-user in order to define the coverage criteria. Each solution that satisfies this rule is considered as a U-CIT testable entity to be covered. What makes this rule special is that UCIT-ASP approach uses this rule both as it is and by deriving more rules from this rule, as will be explained below.

For example, in the example above, if a coverage criterion is defined to cover all 2-way any node orderings in a graph, then the only rule that the end-user needs to add would be the following rule:

```
entity(A, B) :- any_order(A, B).
```

With this given rule, UCIT-ASP automatically computes all the testable entities to be covered for 2-way any node orderings.

3.2 Definition of Valid Test Space

The UCIT-ASP method defines a special rule called `testcase`, which can take as many parameters as needed to define a valid test space. The purpose of this rule is to determine the constraints that a valid test must satisfy.

For the example, as discussed in this section, since there is a path from the start node to the end node in the graph, a valid test case for this example can be defined as follows:

```
testcase :- reaches(S1, S2), start_state(S1), end_state(S2).
```

3.3 Computation of U-CIT Objects

With the given `entity(...)` and `testcase(...)` rules, the U-CIT object computes both the entity space to be covered and the set of valid test cases to be used for this purpose. The next step is to compute the U-CIT object in interest by sampling the valid test case space.

In U-CIT paper (Mercan et al. (2020)), two computation methods (in other word, covering array constructor) have been introduced for this purpose: *generate-and-cover* and *cover-and-generate* methods. The basic logic of generate-and-cover method is to check whether each U-CIT entity is covered by test cases that have already been generated, and in case of the existence of an uncovered U-CIT entity, to generate a new U-CIT test case covering that entity. Cover-and-generate, on the other hand, aims to create a cluster pool by trying to collect the U-CIT entities in the same cluster where the entities in the same cluster can be satisfied together (for example, covered in the same U-CIT test case) so that the combination of all cluster in this pool covers the entire set of testable U-CIT entities.

In our study, we introduced a third computation method which is different from these two computation methods was developed and this method has been implemented in ASP. This method generates a test case to cover the *most* number of entities among the entities that are not yet covered in each step. Iterations continue until there are not testable entities left to cover. Note that while this method tries to optimize the number of entities covered in each step, generate-and-cover method aims to include at least 1 new entity at each step as needed. In other words, it does not deal with the optimization part.

For this purpose, UCIT-ASP first generates `entity_covered(...)` rules to find entities covered by a test case using `entity(...)` rules. For example, for the rule given below,

```
entity(A, B) :- any_order(A, B).
```

the automatically generated corresponding `entity_covered(...)` rule is as follows:

```
entity_covered(A, B) :- entity(A, B), any_order(A, B).
```

This rule means that if `entity(A, B)` is a testable entity to be covered and all entity constraints are satisfied (i.e., `any_order(A, B)` rule is satisfied), then that entity is covered.

Algorithm 1 `entity_covered` ASP rule generation algorithm as pseudocode.

procedure GENERATEENTITYCOVEREDRULE(*entityRule*)
  head, *headArgs* ← *ParseRuleHead*(*entityRule*)
  body, *bodyArgs* ← *ParseRuleBody*(*entityRule*)
  if *headArgs* ≠ *bodyArgs* **then** ▷ returns if entity definition is invalid
    return ""
  entityCoveredHead ← *BuildASPFact*("entity_covered",headArgs)
  entityCoveredBody ← *ConcatWithComma*(head,body)
  entityCoveredRule ← *ConcatRuleSep*(entityCoveredHead,entityCoveredBody)
  return *entityCoveredRule*

Note that automatic generation of the `entity_covered(...)` rule for a given `entity(...)` rule is a mechanical task. This rule has the same number of parameters as the `entity(...)` rule, and the body of this rule (the part after the `:-` symbol) is generated by combining the head of the given `entity(...)` rule (the part before the `:-` symbol) and its body, as illustrated in Algorithm 1.

To ensure the fact that a valid test case is generated at each step, UCIT-ASP uses the following ASP constraint:

```
:- not testcase.
```

In other words, it is ensured that the solution obtained creates a test case. The following optimization directive is also used to ensure that this test case covers the most number of entities that are not covered yet:

```
#maximize{1, A, B : entity_covered(A, B)}.
```

Since here we deal with combinatorial problems, it may not always be practical (or possible) to reach the global optimum in the optimization step. In this case, the ASP solver is run under certain time constraints to find local optimums. If a solution is found within the time constraint for each step, that solution is an optimal solution. If at least one solution is found but the solution space is not completely scanned, the solution is the best among the solutions found in the given time constraint.

If no solution is found at all, all the remaining entities are tried to be covered one-by-one to ensure that this situation is not due to the time limit determined. Note

that covering each entity alone significantly reduces the size of the problem to be solved. When the entity that cannot be covered alone will be marked as invalid, after this step, the invalid entities among the remaining entities will be sorted out and reported. If there are still valid entities among the remaining entities, this indicates that these entities can be covered by increasing the time limit used.

The automatic invalid entity detection functionality provided by UCIT-ASP tool is actually an important feature. Namely, in the studies that we carried out, we have observed that in some cases it is costly to list valid entities, and in such cases, it is reasonable to go for solutions that can also select invalid entities in order to use resources more effectively. Since UCIT-ASP recognizes and eliminates possible invalid entities, this does not cause any negativity in the computation of UCIT-ASP objects.

Once a solution is computed (which actually corresponds to a test case), the corresponding test case is included in the computed U-CIT object. Then, entities covered by this test case are removed from the entity list that contains entities to be covered. Iterations continue until the entity list is empty. All of computed test cases correspond to a U-CIT object that achieves full-coverage under the given coverage criterion.

3.4 Development of UCIT-ASP

We developed UCIT-ASP tool by using clingo (Gebser, Kaminski, Kaufmann & Schaub (2018)) ASP solver, and Python programming language. With the tool developed, test scenarios are defined by using three text files named `system_model.ucit`, `coverage_criterion.ucit` and `test_space.ucit`. These files contain directives and rules for both ASP programs and UCIT-ASP tool.

UCIT-ASP works in two steps. In the first step, all the testable entities to be covered under a given coverage criterion are computed. In the second step, a U-CIT object covering these entities is computed. In the rest of the document, the first step will be referred as *enumeration* step and the second step will be referred as *construction* step.

Among the mentioned files, the `system_model.ucit` file contains general and common definitions of test scenarios. Unless otherwise is stated, the definitions in this

file are used both in the enumeration step and in the construction step. For example, in the example used in this section, the graph modeling definitions of the system under test (Figure 3.2) can be included in the `system_model.ucit` file.

The `coverage_criterion.ucit` file, which is the second file mentioned above, contains the rules that determine the coverage criterion (i.e., `entity(...)` rules). Therefore, the enumeration step is performed by using the definitions in the `system_model.ucit` and `coverage_criterion.ucit` files together. In other words, by solving the ASP rules contained in or generated from both the `system_model.ucit` and `coverage_criterion.ucit` files. For example, the `entity(A, B)` and `any_order(A, B)` rules developed for the graph aforementioned above, are located in the `coverage_criterion.ucit` file.

The third file, `test_space.ucit`, is used to determine the valid test cases space. In other words, this file contains `testcase(...)` rules. Therefore, in the construction step, ASP rules in both `system_model.ucit` and `test_space.ucit` files (or generated from these files) are used together with the entities determined in the enumeration step.

3.4.1 Configuration Parameters

Two main configuration parameters have been implemented in UCIT-ASP system to increase both the run-time performance and scalability of UCIT-ASP. These parameters are named as `max_entities` and `coverage_max_entities` in the rest of the document. The first parameter represents the maximum number of uncovered entities that will be used while computing each test case. When this parameter is not given, UCIT-ASP tries to achieve optimum coverage by using the entire entity list not yet covered. However, since there may be too many entities to be covered in combinatorial problems, using all entities simultaneously in the optimization problem may reduce both run-time performance and scalability. To prevent this, `max_entities` parameter can be used. When this parameter is used, it is used to solve the optimization problem by randomly selecting only the number of entities that have not yet been covered determined by this parameter from the entity list in each step. Therefore, the coverage provided by the computed test case will be limited to the chosen entities. Although this approach may cause an increase in the size of U-CIT objects to be computed, it generally increases computation times and scalability.

In cases where this parameter is used, although the coverage of a test case is computed over the selected set of entities, the test case may actually cover more than these entities. Therefore, the coverage provided by the selected test case is calculated by using the entire entity list not yet covered. However, for this procedure as well, problems may arise in terms of both run-time performance and scalability. Therefore, we developed `coverage_max_entities` parameter to deal with these problems. In cases where this parameter is used, the coverage of a selected test case is performed step-by-step by using a maximum number of different entities determined by the `coverage_max_entities` parameter at each step. The set of entities used during all these steps actually corresponds to the entire set of entities that contains entities which have not yet been covered. Therefore, the coverage provided by the selected test case can be calculated, while increasing the scalability of the system by performing the coverage step-by-step. Note that UCIT-ASP approach uses two parameters instead of a single parameter, since the computation problem of a test case and the coverage calculation of a selected test case are different problems (the second problem is a simpler problem than the first problem).

3.4.2 ASP-Based Modelling Libraries

Within the scope of our studies, we observed that the same or similar models can be used to test very different systems (such as mobile applications and multi-threaded systems). Based on this observation, we developed ASP modeling libraries that contain ASP rules required for the model types that are frequently used in the studies and the ASP directives that can generate these rules parametrically. All functionality provided by these libraries can be used in `system_model.ucit`, `coverage_criterion.ucit` and `test_space.ucit` files as UCIT-ASP directives. UCIT-ASP directives are directives that start with `##` character string and end with `##` character string. The libraries developed within the scope of our studies and the directives provided by these libraries are presented in Table 3.1, 3.2, 3.3 and 3.4. Also, ASP rules generated by these ASP directives are presented in Appendix A.

For example, when the following directive of `graphs` library which has been developed for graph-based models is used,

```
## graphs.graphs.any_order ${'t':2}$ ##
```

`any_order(...)` rule given in Table 3.1 and all other ASP rules completing rules generated by directive to make sense (which are other rules in Figure 3.3) are auto-

directive	description
`any_order ${'t'}$`	Generates ASP code to compute any t-way node orderings.
`consecutive_order ${'t'}$`	Generates ASP code to compute consecutive t-way node orderings.
`nonconsecutive_order ${'t'}$`	Generates ASP code to compute non-consecutive t-way node orderings.
`def_use_pair_def`	Generates ASP code to compute def-use pairs.
`single_def_clear_path_def`	Generates ASP code to determine def-clear-paths in a given path.
`multi_path_def_clear_path_def`	Generates ASP code to determine def-clear-paths in all possible paths.
`any_order_t1_tuple_t2 ${'t1', 't2'}$`	Generates ASP code to compute all testable entities under (t1, t2)-way any ordered path aware coverage criterion.
`consecutive_order_t1_tuple_t2 ${'t1', 't2'}$`	Generates ASP code to compute all testable entities under (t1, t2)-way consecutive ordered path aware coverage criterion.
`nonconsecutive_order_t1_tuple_t2 ${'t1', 't2'}$`	Generates ASP code to compute all testable entities under (t1, t2)-way non-consecutive ordered path aware coverage criterion.

Table 3.1 The list of directives defined in `graphs` library.

directive	description
`minimal_forbidden_tuple ${'tuple'}$`	Generates ASP constraints to describe the combination of minimal invalid parameter values.
`t_tuple ${'t'}$`	Generates ASP code to express t-tuple combinations with their valid parameter values.

Table 3.2 The list of directives defined in `configs` library.

directive	description
`decision_system ${'input'_mode',['file'\|'decisions']}$`	Generates ASP rules for logical expressions in a given decision system.
`bool_expr ${'name', 'expr'}$`	Generates ASP rules for given Boolean expressions.
`bool_var ${'vars'}$`	Generates ASP rules for Boolean variables provided.

Table 3.3 The list of directives defined in `ucit` library.

matically generated by `graphs` library.

The point is that the directives can be parametric, which increases the flexibility and the usability of the approach developed by providing parametrically generation of ASP rules. For example, if the coverage strength $t = 3$ in the directive above, in other words, if the following directive was used,

```
## graphs.graphs.any_order ${‘t’:3}$ ##
```

then `any_order(...)` rule would be defined as:

```
any_order(A, B, C) :- reaches(A, B), reaches(B, C).
```

UCIT-ASP is designed to allow flexible integration of different libraries into the whole system. Adapter design patterns were used for this work. Therefore, UCIT-ASP can be easily expanded with new libraries to be developed.

3.4.3 Non-ASP-Based Modelling Libraries

directive	**description**
`any_order ${'t'}$`	Computes any t-way node orderings by using graphy theory.
`consecutive_order ${'t'}$`	Computes consecutive t-way node orderings by using graphy theory.
`nonconsecutive_order ${'t'}$`	Computes non-consecutive t-way node orderings by using graph theory.
`path_aware_any_order ${'t1', 't2'}$`	Computes all testable entities under (t1, t2)-way any ordered path aware coverage criterion by using graph theory.
`path_aware_consecutive_order ${'t1', 't2'}$`	Computes all testable entities under (t1, t2)-way consecutive ordered path aware coverage criterion by using graph theory.
`path_aware_nonconsecutive_order ${'t1', 't2'}$`	Compute all testable entities under (t1, t2)-way non-consecutive ordered path aware coverage criterion by using graph theory.
`def_use_pairs`	Compute def-use pairs by using graph theory.

Table 3.4 The list of directives defined in `graph_theory` library

In the studies that we carried out, we have observed that in some cases, ASP-based declarative programming approaches may be slower than imperative programming approaches, particularly in the computation of entities (enumeration step) to be covered under a given coverage criterion. For example, in some cases, while analyzing graph-based models, it may be more appropriate to use graph theory instead of ASP-based approaches to solve these problems by using graph theoretical algorithms.

For such cases, non-ASP-based libraries can be developed and integrated with UCIT-ASP tool. Within the scope of our studies, we developed the `graph_theory` library, which computes testable entities by using graph theory algorithms for graph-based models. We developed this library in Python (Table 3.4).

For example, the following parametric directive provided by this library:

```
## graphs.graph_theory.path_aware_any_order ${‘t1’:2, ‘t2’:2}$ ##
```

It states that all testable entities for $(2, 2)$-way path-aware coverage criterion (will be mentioned on Section 4.5) can be computed by using graph theoretical approaches, instead of ASP.

Note that these libraries only take part in the enumeration of testable entities. The computation of U-CIT objects requested is carried out by ASP as handled in Section 3.3.

4. EXPERIMENTS

We have conducted a series of experiments to evaluate the proposed approach in Section 3. In these experiments, firstly we have computed well-known CIT objects as U-CIT objects. Afterward, we have also introduced two novel U-CIT objects by taking advantage of the flexibility of U-CIT. In the rest of this section, in each following subsection related to a U-CIT object, we will introduce the U-CIT object that was used in the experiment and state how we model this U-CIT object in ASP. Then, we will present the results of experiments that have been conducted about the computation of this object. Specifically, we will discuss about standard covering arrays, test case aware covering arrays, decision covering arrays, def-use pair covering arrays, and path aware covering arrays in respectively Section 4.1, 4.2, 4.3, 4.4 and 4.5.

4.1 Computing Standard Covering Arrays as U-CIT Objects

In this section, we will present standard covering array experiments and state how we formulate standard covering arrays by using ASP encoding.

4.1.1 Standard Covering Arrays

The t-way covering arrays (Nie & Leung (2011)), which are frequently used in the Combination Interaction Testing (CIT), takes a configuration space model as an input which is the configuration parameters taking a finite set of values. For a given configuration space model, a t-way standard covering array is a set of configurations constructed to cover at least one combination of relevant parameter values for each

t-way subset of the configuration parameters (Cohen, Dalal, Fredman & Patton (1997)). In this definition, t is called the coverage strength. Once the covering array is computed, the testing of the system under test is performed by testing all configurations in the covering array.

$$p_1 : \{false, true\}$$
$$p_2 : \{false, true\}$$
$$p_3 : \{false, true\}$$

Figure 4.1 A simple configuration space model.

Figure 4.1 provides an example of a simple configuration space model. In this figure, a configuration space model is consisted of three configuration parameters (p_1, p_2, p_3) where each parameter can take binary values ($true$, $false$). Thus, the configuration space model has $2^3 = 8$ different configurations. For example, $<p_1 = false,\ p_2 = false,\ p_3 = true>$ and $<p_1 = false,\ p_2 = true,\ p_3 = true>$ are two different configurations in this configuration space.

$\boldsymbol{p_1}$	$\boldsymbol{p_2}$	$\boldsymbol{p_3}$
true	true	false
true	false	true
false	true	true
false	false	false

Table 4.1 A simple 2-way covering array.

Table 4.1 presents an example of a 2-way covering arrays computed for the configuration space model given in Figure 4.1. Since the covering array in the table is a 2-way covering array, it is possible to find at least one configuration that covers a 2-way combination of the parameter values of p_1, p_2, and p_3 in this array. For example, all parameter value combinations for p_1 and p_2 ($<p_1 = true,\ p_2 = true>$, $<p_1 = true,\ p_2 = false>$, $<p_1 = false,\ p_2 = true>$, $<p_1 = false,\ p_2 = false>$) has been covered at least once. This also applies to parameter combinations (p_1, p_3) and (p_2, p_3).

Configuration space models may contain system constraints to express both invalid configurations and invalid combinations of configuration parameter values (Jia, Cohen, Harman & Petke (2015); Yamada, Biere, Artho, Kitamura & Choi (2016)). For example, if the following constraints, $(p_2 = true) \implies (p3 = true)$ (if p_2 is true, p_3 must be true) and $\neg(p_1 = true \wedge p_3 = false)$ (combination of $p_1 = true$ and $p_3 = false$ is invalid) are given, the 2-way covering array in Table 4.2 would be obtained. Note that each configuration in the computed 2-way covering array satisfies all the system constraints, and any 2-way parameter value combination which does not satisfy the constraints is not included in this array.

p_1	p_2	p_3
true	true	true
true	false	true
false	true	true
false	false	false

Table 4.2 An example of 2-way covering array with system constraints.

4.1.2 U-CIT Formulation

This section presents the ASP formulation of standard covering arrays. As discussed in Section 3.4, test scenarios in the UCIT-ASP method are defined using three different input files (`system_model.ucit`, `coverage_criterion.ucit` and `test_space.ucit`).

`system_model.ucit`: In this study, the configuration parameters in the configuration space model and the finite values of these parameters are defined in the `system_model.ucit` file. For this purpose, the `option(...)` rule is defined. In this ASP rule, the first parameter represents the name of the configuration parameter, while the second parameter represents the set of values that the defined parameter can take. For example, the `system_model.ucit` file in Figure 4.2 models a system with four configuration parameters (o1, o2, o3 and o4), each of parameters can take a binary value (0 or 1).

```
% Each configuration parameters takes binary values (0 and 1)
option(o1, 0..1).
option(o2, 0..1).
option(o3, 0..1).
option(o4, 0..1).

## configs.configs.minimal_forbidden_tuple ${'tuple':{'o1':1, 'o2':1}}$ ##
## configs.configs.minimal_forbidden_tuple ${'tuple':{'o3':1, 'o4':1}}$ ##

## configs.configs.t_tuple ${'t':2}$ ##
```

Figure 4.2 An example `system_model.ucit` file

```
% Entity definition
entity(O1, V1, O2, V2) :- t_tuple(O1, V1, O2, V2).
```

Figure 4.3 An example `coverage_criterion.ucit` file

Standard covering arrays are concerned with the interaction between parame-

```
% Test case definition
testcase.
```

Figure 4.4 An example `test_space.ucit` file

ters. However, certain combinations of parameter values might be invalid. The `minimal_forbidden_tuple` directive in the `configs` library that we developed developed in UCIT-ASP can be used to specify invalid parameter combinations. For example, in Figure 4.2

```
## configs.configs.minimal_forbidden_tuple ${‘tuple’:{‘o1’:1,‘o2’:1}}$ ##
```

directive states that ($o1 = 1$, $o2 = 1$) combination is an invalid combination. This directive automatically generates the following ASP rules to ensure that the specified system constraint is met:

```
minimal_forbidden_tuple(o1, 1, o2, 1).
```

`coverage_criterion.ucit`: In Figure 4.3, a sample `coverage_criterion.ucit` file that can be used for the testing scenario mentioned in this section is illustrated.

The definition of coverage strength t is also illustrated by using the parametric `t_tuple` directive defined in the `configs` library developed:

```
## configs.configs.t_tuple ${‘t’:2}$ ##
```

This directive automatically generates the rule `t_tuple(O1, V1, O2, V2)` to find all valid 2-way parameter value combinations as follows:

```
t_tuple(O1, V1, O2, V2) :- O1 < O2,
    option(O1, V1), option(O2, V2),
    not minimal_forbidden_tuple(O1, V1),
    not minimal_forbidden_tuple(O2, V2),
    not minimal_forbidden_tuple(O1, V1, O2, V2).
```

Note that the `t_tuple(O1, V1, O2, V2)` rule represents a valid 2-way combination (`O1=V1`, `O2=V2`). As seen from the following rule in Figure 4.3,

```
entity(O1, V1, O2, V2) :- t_tuple(O1, V1, O2, V2).
```

for the coverage criterion, in this example, it is stated that all combinations of 2-way parameter values are covered. In fact, it corresponds to the computation of a 2-way standard covering array.

`test_space.ucit`: Finally, Figure 4.4 presents an example `test_space.ucit` file.

As covered in Section 3.2, the purpose of this file is to define the valid test space using `testcase` rule. However, since a valid test case for the scenario discussed in this section is to generate a valid configuration in which each defined configuration parameter takes a value, and this constraint is already provided by the rules in the `system_model.ucit` file given in Figure 4.2. Therefore, `testcase` rule is defined empty without needing for an extra definition as follow:

```
testcase.
```

4.1.3 Experiment

In this section, the related experiment information will be provided on computing standard covering arrays as U-CIT objects. In the rest of the section, we will present respectively our experiment setup, evaluation framework, operational framework, data and analysis of experiment results, and finally, the discussion of experiment results.

4.1.3.1 Setup

To evaluate the proposed approach on standard covering array generation, the configuration space models of Apache and MySQL web servers have been used as experiment models (Yilmaz (2013)). Table 4.3 shows the model details for these applications. More details about the experiment models can found in Section 4.2.3.1.

	Apache	MySQL
parameter count	13	12
2-way testable entities	311	333
3-way testable entities	2261	2476

Table 4.3 The information about experiment models.

4.1.3.2 Evaluation Framework

In this section, we present our evaluation framework to demonstrate our evaluation metrics for the experiments. To evaluate the experiments, we use covering array sizes (also referred as U-CIT object size), testable entity enumeration times, covering array construction times and covering array size improvement percentages in comparison to other covering array constructors as evaluation metrics.

Covering array size represents the number of generated configuration tuples and indicates the effectiveness of the approach. In comparisons, approaches generating covering arrays with smaller covering array sizes indicate the fact that testable entities are covered with less number of test cases vice versa. Therefore, these approaches are considered more effective than others. Testable entity enumeration time states the elapsed time during entity enumeration. Covering array construction time states the elapsed time during the construction of a covering array. While comparing covering array constructors, a constructor with smaller values of these three metrics is considered as performing better. Also, covering array size improvement percentage is another measurement metric used in this study which expresses how much the array size is improved by using our constructor with respect to others.

4.1.3.3 Operational Framework

Unless otherwise stated, all experiments in this study were repeated 5 times on an Intel Xeon CPU 2.30GHz Google Cloud machines with 64-bit Ubuntu 18.10 operating system and 4Gb RAM. Clingo (v4.5.4) has been used as the ASP solver during experiments. For each test case generation phase, 30 seconds time limit has been set to the ASP solver to find a satisfiable solution, in other words a test case.

4.1.3.4 Data and Analysis

A series of experiments have been performed to evaluate the proposed approach on standard covering array generation. In the experiments, 2-way and 3-way covering arrays have been generated for the experiment models mention in Section 4.1.3.1. We used *one-test-case-a-time* constructor for this study to represent our approach. Also, in order to compare our approach, we produced covering arrays with well-known covering array constructors (Jenny and ACTS) in the literature (Jenkins (2005); Yu, Lei, Kacker & Kuhn (2013)). Table 4.4 illustrates the experiments

results performed in the scope this study.

						time (seconds)	
model	**parameters**	**t**	**entities**	**approach**	**size**	**enumeration**	**construction**
apache	13	2	311	ucit-asp	9	0.33	1.00
				jenny	10	-	0.01
				acts	11	-	0.34
		3	2261	ucit-asp	21	0.00	13.66
				jenny	21	-	0.01
				acts	24	-	0.36
mysql	12	2	333	ucit-asp	13	0.00	1.00
				jenny	14	-	0.01
				acts	9	-	0.37
		3	2476	ucit-asp	35	0.33	17.66
				jenny	39	-	0.01
				acts	22	-	0.37

Table 4.4 The comparison of UCIT-ASP approach with the standard covering array constructors defined in the literature.

In these experiments, we observed that coverage strength has an important impact on the number of testable entities. While $t = 2$, testable entity counts are 311 and 333; and when $t = 3$ testable entity counts are 2261 and 2476, for Apache and MySQL models respectively. Also, another observation is that our constructor generated smaller covering arrays in comparison with Jenny for all experiment setups. Even in comparison to ACTS, for Apache model, our constructor generates smaller U-CIT objects than ACTS. Table 4.5 shows the covering array size improvements of our constructor in comparison with Jenny and ACTS. However, UCIT-ASP severely suffers in the covering array construction time.

			size improvement (%)	
model	**parameters**	**t**	**vs jenny**	**vs acts**
apache	13	2	10.00	18.18
		3	0.00	12.50
mysql	12	2	7.14	-44.44
		3	1.00	-59.09

Table 4.5 The covering array size improvement of one-test-case-at-a-time constructor in comparison to Jenny and ACTS.

4.1.3.5 Discussion

The aim of our approach is to support scenarios that existing constructors cannot support, rather than replacing existing combinatorial object constructors. In fact,

the aim of the study, in this section, is to demonstrate the flexibility of this method by showing that the developed approach can also compute frequently used standard covering arrays.

Although UCIT-ASP approach has computed CIT objects in more computation time than other approaches, it has generated smaller covering arrays than other approaches in most experiments. Particularly, we observed that up to 18.18% size reductions in the covering arrays generated for the Apache model. The reason for this is can be considered as that Apache model have the smaller value set for each configuration parameter and fewer system constraints than MySQL model which allows the ASP solver to optimize the problems in depth.

4.2 Computing Test Case Aware Covering Arrays as U-CIT objects

In this section, test case aware covering arrays (Yilmaz (2013)) will be introduced and also we will explain how they are modeled as U-CIT objects in ASP through examples.

4.2.1 Test Case Aware Covering Arrays

Unlike standard covering arrays (Section 4.1), test case aware covering arrays compute the covering arrays by considering the test case specific constraints (Yilmaz (2013)). Standard covering arrays ignoring the test case constraints might suffer from masking effects of parameters as a result. Hence, all combinations of parameter values that are required to be tested may not be tested at all. Thus, test case aware covering arrays significantly increase both the applicability and the effectiveness of the CIT approaches by allowing all combinations required to be tested for each test case.

To understand the test case aware covering arrays, the sample configuration space model given in Figure 4.1 can be extended. The difference of our application scenario from the example is that each valid configuration does not represent a test case as it is. In other words, in this study, systems under test have test cases that are designed to run on selected configurations and these test cases have their own specific

constraints.

For example, let the system defined in Figure 4.1 have two test cases: $test_1$ and $test_2$. Let these test cases have the following constraints: $test_1$ cannot be run when $p_2 = true$ and $p_3 = true$, and $test_2$ test case cannot be run when $p_3 = false$. Note that the test case, which does not meet its own test case constraint, will not run on any configuration. For example, $test_1$ will not run in the first configuration in Table 4.2 ($<p_1 = true$, $p_2 = true$, $p_3 = true>$). In this case, another combination of 2-way parameter values ($p_1 = true$, $p_2 = true$), which is a valid configuration for the test case ($test_1$), cannot be tested by $test_1$. In other words, this configuration will be masked (Yilmaz et al. (2014)). The purpose of test case aware covering arrays is to eliminate the masking effect of standard covering arrays by allowing each test case to be run on each valid t-way combination of parameter values.

Therefore, test case aware covering arrays take a configuration space model as input that includes configuration parameters, the set of values that these parameters can take, system constraints, a list of test cases, and test case specific constraints (if any). For a given configuration space model and coverage strength (t), a t-way test case aware covering array is a set of configurations in which each configuration is associated with a set of test cases. In this structure, the set of test cases associated with a configuration refers to the test cases scheduled to be run in that configuration.

A t-way test case aware covering array is computed to provide the following properties: 1) all selected configurations meet the system constraints, 2) none of test cases are scheduled to run in configurations that do not meet their specific constraints, and 3) for each test case, all of the configurations, which the test case is scheduled for, constitute a t-way standard covering array containing all valid t-way combinations of parameter values for the test case at least once. Table 4.6 illustrates a computed test case aware covering array for the example considered above.

p_1	p_2	p_3	t_1	t_2
true	true	true	false	true
true	false	true	true	true
false	true	true	false	true
false	false	false	true	false
false	false	true	true	true
false	true	false	true	false
true	true	false	true	false

Table 4.6 An example of a 2-way test case aware covering array.

4.2.2 U-CIT Formulation

In this section, we will show how test case aware covering arrays are formulated by using UCIT-ASP method. Test case aware covering arrays extend the coverage criterion of standard covering arrays by adding extra test specific constraints. Therefore, system models of both CIT objects are similar to each other.

```
% Parameters and their value set
option(o1, 0..1).
option(o2, 0..3).
option(o3, 0..1).
option(o4, 0..2).

% System constraints
## configs.configs.minimal_forbidden_tuple ${'tuple':{'o3':1,'o4':1}}$ ##

% 2-way parameter value combinations
## configs.configs.t_tuple ${'t':2}$ ##

% Test cases
test(t1).
test(t2).

% Test case specific constraints
test_specific_minimal_forbidden_tuple(t1, o1, 1).
test_specific_minimal_forbidden_tuple(t2, o2, 2).

% Valid 2-way parameter combinations for test cases
schedule_test(T, O1, V1, O2, V2) :- test(T), O1 < O2,
   option(O1, V1), option(O2, V2),
   not test_specific_minimal_forbidden_tuple(T, O1, V1),
   not test_specific_minimal_forbidden_tuple(T, O2, V2),
   not test_specific_minimal_forbidden_tuple(T, O1, V1, O2, V2).
```

Figure 4.5 An example `system_model.ucit` file.

```
% Coverage criterion
entity(T, O1, V1, O2, V2) :- O1 < O2, test(T),
   t_tuple(O1, V1, O2, V2),
   schedule_test(T, O1, V1, O2, V2).
```

Figure 4.6 An example `coverage_criterion.ucit` file.

`system_model.ucit`: As in standard covering arrays, in this scenario, all parameters in the system are defined in the `system_model.ucit` file by using the `option(...)` rule (Section 4.1.2). In addition, system-wide constraints (i.e. constraints that must

```
% Test case
testcase.
```

Figure 4.7 An example `test_space.ucit` file.

be applied for all test cases) are expressed by using `minimal_forbidden_tuple` directive provided in the `configs` library (Section 3.4.2).

Moreover, two new ASP rules are used to make specific definitions for test case aware covering arrays. These rules are `test(...)` and `test_specific_minimal_forbidden_tuple(...)`. While the first rule defines test cases, the second defines test case specific constraints (i.e. combinations of parameter values for which test cases will not run).

In Figure 4.5, a system with two test cases, test case specific constraints for these test cases and a system constraint are modeled to be used in this study. For example,

```
test_specific_minimal_forbidden_tuple(t1, o1, 1).
```

rule states that $t1$ test cannot run when $o1 = 1$

```
test_specific_minimal_forbidden_tuple(t2, o2, 2).
```

The rule also states that the $t2$ test cannot run when $o2 = 2$. Also, in this file,

```
schedule_test(T, O1, V1, O2, V2)
    :- test(T), O1 < O2,
    option(O1, V1), option(O2, V2),
    not test_specific_minimal_forbidden_tuple(T, O1, V1),
    not test_specific_minimal_forbidden_tuple(T, O2, V2),
    not test_specific_minimal_forbidden_tuple(T, O1, V1, O2, V2).
```

rule specifies that in which configurations the test case will run and in which configurations it will not run, by determining the valid combination of 2-tuple parameter values (`O1=V1`, `O2=V2`) for each test case `T`. Note that the definitions here may vary depending on the definitions of invalid combinations or coverage strength to be achieved. However, it is clear how to adapt in such cases.

`coverage_criterion.ucit`: The coverage criterion specified in the `coverage_criterion.ucit` file in Figure 4.6,

```
entity(T, O1, V1, O2, V2) :- O1 < O2, test(T),
    t_tuple(O1, V1, O2, V2),
    schedule_test(T, O1, V1, O2, V2).
```

computes a 2-way test case aware covering array by expressing that every combination of valid 2-tuple parameter values for each test case must be covered.

`test_space.ucit`: As in the previous section (Section 4.1.2), since definitions provided in `system_model.ucit` file already express what a test case means, `testcase` rule defining test cases is defined as an empty rule (Figure 4.7):

```
testcase.
```

4.2.3 Experiment

In this section, the related experiment information will be provided on computing test case aware covering arrays as U-CIT objects. In the rest of the section, we will present respectively our experiment setup, evaluation framework, operational framework, data and analysis of experiment results and finally, the discussion of experiment results.

4.2.3.1 Setup

A series of experiments have been carried out to evaluate our approach on test case covering array generation. In these experiments, the same Apache and MySQL web servers in Section 4.1.3.1 (Yilmaz (2013)) were used as experimental models with test cases and extra test case specific constraints. While Table 4.7 and Table 4.8 shows the configuration space model and the test case specific constraints for MySQL model respectively, Table 4.9 and Table 4.10 present the same model details for Apache model.

4.2.3.2 Evaluation Framework

In this study, the same evaluation framework defined in Section 4.1.3.2 has been used to evaluate the proposed approach for test case aware covering arrays.

option	settings
log-format	{row, statement, mixed}
sql-mode	{strict, traditional, ansi}
ext-charsets	{disable, complex, all}
innodb	{enable, disable}
libedit	{enable, disable}
log-bin	{enable, disable}
readline	{enable, disable}
ndbcluster	{enable, disable}
ssl	{enable, disable}
archive	{enable, disable}
blockhole	{enable, disable}
federated	{enable, disable}
system-wide constraint	
ssl=disable[1] ∧ (libedit=enable → readline=disable)	

Table 4.7 Traditional configuration space model used in the experiments for MySQL (taken from Yilmaz (2013)).

cluster idx	# of tests	test case-specific constraint
1	86	log-bin=enable ∧ sql-mode≠ansi
2	60	ndbcluster=enable
3	33	innodb=enable
4	28	log-format≠row ∧ log-bin=enable ∧ sql-mode≠ansi
5	22	sql-mode≠ansi
6	18	ext-charsets≠disable ∧ sql-mode≠ansi
7	17	log-format≠statement ∧ log-bin=enable ∧ ndbcluster=enable
8	17	innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
9	16	log-bin=enable ∧ ndbcluster=enable
10	6	log-format≠row ∧ innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
11	4	log-format≠row ∧ ext-charsets≠disable ∧ log-bin=enable ∧ sql-mode≠ansi
12	4	federated=enable ∧ log-bin=enable ∧ sql-mode≠ansi
13	4	innodb=enable ∧ sql-mode≠ansi
14	4	ndbcluster=enable ∧ sql-mode≠ansi
15	2	log-format≠statement ∧ innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
16	2	blackhole=enable ∧ log-bin=enable ∧ ndbcluster=enable
17	1	archive=enable ∧ log-format≠row ∧ log-bin=enable ∧ sql-mode≠ansi
18	1	federated=enable ∧ innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
19	1	log-format≠row ∧ blackhole=enable ∧ log-bin=enable ∧ sql-mode≠ansi
20	1	log-format≠statement ∧ log-bin=enable ∧ ndbcluster=enable ∧ sql-mode≠ansi
21	1	ext-charsets≠disable ∧ log-bin=enable ∧ sql-mode≠ansi
22	1	log-bin=enable ∧ ndbcluster=enable ∧ sql-mode≠ansi
23	1	log-format≠row ∧ log-bin=enable ∧ ndbcluster=enable
24	1	ext-charsets≠disable ∧ innodb=enable ∧ sql-mode≠ansi
25	1	innodb=enable ∧ log-bin=enable ∧ ndbcluster=enable
26	1	innodb=enable ∧ ndbcluster=enable
27	1	archive=enable ∧ innodb=enable
28	1	archive=enable
29	1	log-bin=enable
30	1	ext-charsets≠all

Table 4.8 Test case-specific constraints used in the experiments for MySQL (taken from Yilmaz (2013)).

option	settings
case-filter	{enable, disable}
ssl	{enable, disable}
dav	{enable, disable}
echo	{enable, disable}
rewrite	{enable, disable}
case-filter-in	{enable, disable}
bucketeer	{enable, disable}
info	{enable, disable}
headers	{enable, disable}
vhost-alias	{enable, disable}
cgi	{enable, disable}
proxy-http	{enable, disable}
proxy	{enable, disable}
system-wide constraint	
proxy-http = enable → proxy=enable	

Table 4.9 Traditional configuration space model used in the experiments for Apache (taken from Yilmaz (2013)).

cluster idx	# of tests	test case-specific constraint
1	172	ssl=enable ∧ proxy-http=enable
2	74	ssl=enable
3	26	rewrite=enable
4	22	headers=enable
5	21	proxy=enable
6	16	dav=enable
7	11	case-filter=enable
8	8	vhost-alias=enable
9	7	proxy-http=enable
10	5	proxy=enable∧rewrite=enable ∧ cgi=enable
11	4	echo=enable
12	3	ssl=enable ∧ headers=enable
13	2	rewrite=enable ∧ proxy=enable
14	2	ssl=enable ∧ case-filter-in=enable
15	2	case-filter-in=enable
16	2	bucketeer=enable
17	1	info=enable

Table 4.10 Test case-specific constraints used in the experiments for Apache (taken from Yilmaz (2013)).

4.2.3.3 Operational Framework

In this study, the same operational framework defined in Section 4.1.3.3 has been used to carry out the experiments.

4.2.3.4 Data and Analysis

Table 4.11 presents the experiment results of computing 2-way and 3-way test case covering arrays for Apache and MySQL web servers. Table 4.12 presents covering size improvement ratios obtained by using UCIT-ASP approach. As stated in the standard covering arrays (Section 4.1.3.4), covering array sizes and construction times increase as entity sizes increase. As seen in Table 4.12, UCIT-ASP has outperformed tse-algo introduced in Yilmaz (2013) in terms of array size improvements in Apache experiments. We observed covering array size improvement in Apache experiments up to 5.77%.

					time (seconds)	
model	**t**	**entities**	**approach**	**size**	**enumeration**	**construction**
apache	2	4994	ucit-asp	25.2	0.2	237.2
			tse-algo	25.5	-	60.0
	3	35174	ucit-asp	62.1	2.2	9624.5
			tse-algo	65.9	-	420.0
mysql	2	9833	ucit-asp	42.3	0.0	386.6
			tse-algo	42.2	-	60.0
	3	72402	ucit-asp	133.4	3.6	37217.5
			tse-algo	130.0	-	1020.0

Table 4.11 The comparison of UCIT-ASP approach with approaches (Yilmaz (2013)) defined in the literature.

model	**t**	**parameters**	**system constraints**	**tests**	**size improvement (%)**
apache	2	13	1	17	1.18
	3				5.77
mysql	2	12	5	30	-0.24
	3				-2.62

Table 4.12 The array size improvement comparison of UCIT-ASP approach with the test case covering array construction algorithm proposed in the literature (Yilmaz (2013)).

4.2.3.5 Discussion

As discussed in the previous section (Section 4.1.3.5), the purpose of this study is not to compare the our approach with the specialized approaches. The main purpose of this study is to demonstrate the flexibility of this approach by showing that the approach developed can compute a different CIT object published in the literature. Even though UCIT-ASP approach requires more time to compute CIT objects, it has decreased the size of CIT objects up to 5.77% for the Apache model in comparison with the other method in the literature.

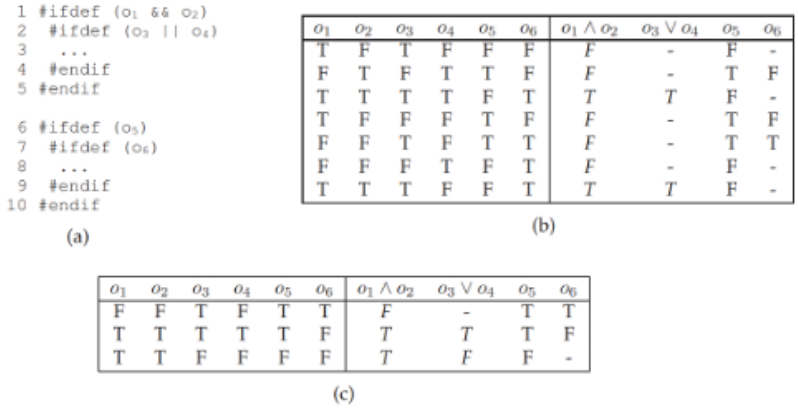
4.3 Computing Decision Covering Arrays as U-CIT Objects

In this section, first of all, we will introduce decision covering arrays and explain how they are formulated as U-CIT objects and computed with UCIT-ASP. In addition, experiments results obtained from the experiments to measure the efficiency and

effectiveness of UCIT-ASP in the computation of decision covering arrays will be provided.

4.3.1 Decision Covering Arrays

In this study, we are interested in compile-time configuration parameters declared by using preprocessor directives such as `#ifdef` and `#ifndef` preprocessor directives used in C and C++. Figure 4.8 shows a hypothetical system with 6 compile-time configuration parameters $(o_1, o_2, \ldots, o_6)$, where each parameter can take binary values (T: $true$ or F: $false$). In the rest of the document, `#ifdef`, `#ifndef`, or similar conditional branching directives will be referred as an if-then-else directive. Since these directives consist of compile-time parameters, they can be easily checked externally during the compilation and build phase of the system.



```
1 #ifdef (o1 && o2)
2  #ifdef (o3 || o4)
3   ...
4  #endif
5 #endif

6 #ifdef (o5)
7  #ifdef (o6)
8   ...
9  #endif
10 #endif
```

(a)

o_1	o_2	o_3	o_4	o_5	o_6	$o_1 \wedge o_2$	$o_3 \vee o_4$	o_5	o_6
T	F	T	F	F	F	*F*	-	F	-
F	T	F	T	T	F	*F*	-	T	F
T	T	T	T	F	T	*T*	*T*	F	-
T	F	F	F	T	F	*F*	-	T	F
F	F	T	F	T	T	*F*	-	T	T
F	F	F	T	F	T	*F*	-	F	-
T	T	T	F	F	T	*T*	*T*	F	-

(b)

o_1	o_2	o_3	o_4	o_5	o_6	$o_1 \wedge o_2$	$o_3 \vee o_4$	o_5	o_6
F	F	T	F	T	T	*F*	-	T	T
T	T	T	T	T	F	*T*	*T*	T	F
T	T	F	F	F	F	*T*	*F*	F	-

(c)

Figure 4.8 (a) An example of preprocessor directives with 6 compile-time configuration parameters, (b) An example of 2-way standard covering array generated for this system, and (c) An example of test cases obtaining full coverage under decision coverage criterion.

Moreover, these if-then-else directives types express how the configuration parameters interact with each other. Since the decisions to be taken might change with respect to these interactions and might change the way of the system works, these interactions need to be tested. Therefore, decision coverage (DC) criterion takes its place as a structural test adequacy criterion that software developers can use in such scenarios. In order to achieve full coverage with DC criterion, both $true$ and $false$ results must be obtained for all decisions (such as $o_1 \wedge o_2$ and $o_3 \vee o_4$ in Figure 4.8.a).

Since standard covering arrays do not take into account the interaction between configuration parameters, in cases where there are nested parameters, they usually

either cannot achieve full coverage or require too many test cases to achieve full coverage (Javeed (2015); Javeed & Yilmaz (2015)). For example, for the scenario given in Figure 4.8.a, we want to obtain full coverage under DC. For this reason, let's use a 2-way covering array that is partially given in 4.8.b. In this case, the last 4 columns in 4.8.b show the outputs of the decisions taken: T ($true$) and F ($false$) indicate the Boolean outcomes, while '$-$' indicates that the relevant conditional statement cannot be reached due to other conditional statements. For example, the decision $(o_3 \vee o_4)$ in the first line cannot be reached because the $(o_1 \wedge o_2)$ condition cannot be satisfied, and therefore this decision cannot be tested by $(o_1 = T, o_2 = F, o_3 = T, o_4 = F, o_5 = F, o_6 = F)$ configuration.

While for if-then-else directives between lines 1 and 5, it can achieve only 75% coverage, the 2-way covering array in 4.8.b achieves full coverage under DC coverage criterion for the if-then-else directives between lines 6 and 10 in 4.8.a.

UCIT-ASP method takes the source code of the system to be tested, a coverage strength t, and a structural coverage criterion as an input. First of all, each if-then-else directive not included in another if-then-else directive in the source code is defined as a virtual configuration parameter. Then, under the structural coverage criterion requested for a given virtual configuration parameter, each condition that must be satisfied to achieve full coverage is expressed as a virtual value that the virtual configuration parameter can take.

Finally, configurations (test cases) are generated to include valid combinations of t virtual parameter values. The fewer configurations needed for full coverage, the more effective the proposed approach will be.

In the rest of the document, the DC criterion will be used as the structural coverage criterion without compromising the generality of the proposed method. However, the proposed approach is easily suitable to use with other structural coverage criteria such as condition coverage and MC/DC coverage (Yu & Lau (2006)).

Definition 1 *A virtual configuration parameter (or virtual parameter vo_i) is an if-then-else directive that is not nested within another if-then-else directive.*

For example, the system in 4.8.a has two virtual parameters: while vo_1 represents the if-then-else directive between lines 1 and 5, and vo_2 represents the if-then-else directive between lines 6 and 10.

Definition 2 *Every possible outcome of all decisions in the if-then-else directive corresponding to a given virtual configuration parameter is a virtual value for that virtual configuration parameter. Note that each virtual value must be expressed as a*

constraint in U-CIT.

Therefore, when all virtual values of a given virtual configuration parameter are covered, full coverage is obtained under the DC criterion for the relevant if-then-else directive.

For the example in 4.8, vo_1 has four virtual values: $\{(o_1 \wedge o_2), \neg(o_1 \wedge o_2), (o_1 \wedge o_2) \wedge (o_3 \vee o_4), (o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)\}$. The first two virtual values are defined to cover $true$ and $false$ values in the branches formed by $(o_1 \wedge o_2)$ decision, and the next two virtual values are for $true$ and $false$ branches in the decision $(o_3 \vee o_4)$, which is protected by $(o1 \wedge o_2)$ condition. Similarly, the vo_2 parameter has four virtual values: $\{o_5, \neg o_5, (o_5 \wedge o_6), (o_5 \wedge \neg o_6)\}$.

For a given source code of a software system, not all combinations of virtual parameter values may be valid due to conflicts that may arise because of using the actual configuration parameters in more than one place. In these cases, since each virtual value is expressed as an U-CIT constraint, it can be decided whether a given virtual value is valid or not, by looking at whether the corresponding constraint can be solved. In other words, a virtual value will only be valid if the corresponding constraint can be satisfied. In the rest of the document, the term "virtual value" will be used to refer to valid virtual values.

Definition 3 *A t-combination is the constraints corresponding to the virtual values obtained from t different virtual parameters combined with the logical AND operator.*

A t-combination would be invalid if the corresponding constraint with virtual values is not satisfied. In the rest of the document, the term "t-combination" will be used to denote valid t-combinations.

Notice that each t-combination represents an interaction to be tested. If we go back to the previous example, when $t = 2$, some 2-way combinations for vo_1 and vo_2 are as follows: $(o_1 \wedge o_2) \wedge (o_5)$ which tests the interactions between the $true$ branches of decisions on lines 1 and 6 and $((o_1 \wedge o_2) \wedge \neg(o_3 \vee o_4)) \wedge (o_6)$ tests the interaction of $false$ branch on line 2 and $true$ branch on line 7.

Definition 4 *For a given set of virtual configuration parameters, a t-way structural coverage criterion ($K_{structural}$) marks the virtual values taken by the parameters in this set to include all valid t-way combinations for t coverage strength.*

If we go back to the previous example, for $t = 2$, it marks the $K_{structural}$ criterion to include a total of $4x4 = 16$ 2-way combinations for the virtual parameters vo_1 and vo_2.

Definition 5 *A t-way structural U-CIT object is a set configuration consisting of real configuration parameters, including all t-combinations marked by the $K_{structural}$ criterion for the given virtual configuration parameters, the virtual values these parameters take, and the coverage strength t.*

In this context, to include a t-combination for a real system configuration, the constraint corresponding to the t-combination must be satisfied with the configuration in interest.

Note that if the coverage strength t for $K_{structural}$ is 1, all the virtual values of all virtual parameters would be covered. In other words, covering all valid 1-way combinations guarantees full coverage under the given structural coverage criterion. As a result, the 1-way structural U-CIT object corresponds to CIT objects introduced in the (Javeed & Yilmaz (2015)) study. Therefore, our study is a study showing the expressiveness strength of the U-CIT approach.

Although the 1-way structural U-CIT objects test the interactions of the configuration parameters within the if-then-else directives, they do not take into account the interactions between the if-then-else directives that are not structurally related to each other. By considering the 1-way structural U-CIT object given in 4.8.c as an example, although this object provides full coverage under DC criterion, it does not test some interactions between independent if-then-else directives. For example, the interactions of *true* branches of the decisions $o_1 \wedge o_2$ (line 1) and o_6 (line 7) cannot be tested by this given structural object. Therefore, we also compute 2-way decision covering arrays to test interactions of 2-way if-then-else directives, in other words 2-way combinations of decision outcomes.

4.3.2 U-CIT Formulation

In order to compute U-CIT objects, first of all, logical decisions must be defined in UCIT-ASP system. Two new directives have been developed in UCIT-ASP for this purpose: `ucit.ucit.bool_var` and `ucit.ucit.bool_expr`. While the first directive is used to define a Boolean variable, the second is used to define Boolean expressions that use declared variables. Note that these directives parametrically generate the necessary ASP rules, as discussed in the previous sections. Thus, there is no need to develop another ASP directive to achieve the same functionality. ASP rules corresponding to these directives can be used directly in UCIT-ASP system. The aim of developing these directives within the scope of the study is to reduce the

burden on end-users as much as possible by creating ready-made ASP libraries for frequently used model elements based on our development experience.

For example, the following directive defines three Boolean variables: `a`, `b`, and `c`.

```
## ucit.ucit.bool_var ${‘vars’:[‘a’,‘b’,‘c’]}$ ##
```

This directive automatically generates the following ASP directives to express that each variable can take either *true* or $false$ values:

```
1 {parameter(global, a, true); parameter(global, a, false)} 1.
1 {parameter(global, b, true); parameter(global, b, false)} 1.
1 {parameter(global, c, true); parameter(global, c, false)} 1.
```

The following directive defines a Boolean expression `~((a&~b)|c)` for `falseBranch` by using these variables:

```
## ucit.ucit.bool_expr ${‘name’:‘falseBranch’,‘expr’:‘~((a&~b)|c)’}$ ##
```

This directive converts a given Boolean expression to Disjunctive Normal Form (DNF) (so that it can be formulated in ASP) and generates an ASP rule for each element in this form. For example, the directive above produces the following rules:

```
% falseBranch : orig = ~((a & ~b) | c)
% falseBranch : dnf  = Or(And(~a, ~c), And(b, ~c))
falseBranch :- parameter(global, a, false), parameter(global, c, false).
falseBranch :- parameter(global, b, true), parameter(global, c, false).
```

```
% Variable definitions
## ucit.ucit.bool_var ${‘vars’ : [‘a’, ‘b’, ‘c’]}$ ##

% Expression results
## ucit.ucit.bool_expr ${‘name’:‘trueBranch’, ‘expr’:‘(a&~b)|c’}$ ##
## ucit.ucit.bool_expr ${‘name’:‘falseBranch’, ‘expr’:‘~((a&~b)|c)’}$ ##
```

Figure 4.9 An example `system_model.ucit` file.

```
% Entity definitions
entity(e1) :- trueBranch
entity(e1) :- falseBranch
```

Figure 4.10 An example `coverage_criterion.ucit` file.

```
% Test case definition
testcase.
```

Figure 4.11 An example `test_space.ucit` file.

`system_model.ucit`: To compute UCIT objects, `system_model.ucit` file contains Boolean variables used in the system and decisions using these variables. Both satisfied and not satisfied outcomes of decisions also are included in the file. In the example file `system_model.ucit` given in Figure 4.9, both outcomes of `((a&~b)|c)` decision (in other words, when this decision is satisfied or not) are defined as two Boolean expressions, as `trueBranch` and `falseBranch`.

`coverage_criterion.ucit`: To compute decision covering U-CIT objects under structural coverage criterion, it is required to cover all outcomes of each decision. As can be seen in the sample `coverage_criterion.ucit` file in Figure 4.10, this is achieved by showing that all relevant Boolean expressions defined must be covered.

`test_space.ucit`: Since there is no extra constraint to define the valid test space, `testcase` rule is defined as an empty rule, as seen in the sample `test_space.ucit` file given in Figure 4.11.

4.3.3 Experiment

In this section, the related experiment information will be provided on computing decision covering arrays as U-CIT objects. In the rest of the section, we will present respectively our experiment setup, evaluation framework, operational framework, data and analysis of experiment results, and finally, the discussion of experiment results.

4.3.3.1 Setup

A series of experiments have been carried out to evaluate the proposed approach. In these experiments, 12 experiment models obtained from real software systems have been used (Javeed & Yilmaz (2015)). Each application model has binary compile-time configuration parameters built using preprocessor directives. Since no

constraints on the configuration parameters are known, all possible combinations of parameter values are assumed to be valid.

4.3.3.2 Evaluation Framework

In this study, the same evaluation framework defined in Section 4.1.3.2 has been used to evaluate the proposed approach for decision covering arrays.

4.3.3.3 Operational Framework

Unless otherwise is stated, in this study, all the experiments were repeated 5 times and carried out on Google Cloud using Intel Xeon CPU 2.30GHz machines with 4Gb of RAM, running 64-bit Ubuntu 18.04 as the operating system. The time limit configuration parameter of UCIT-ASP, which is used to limit test case generation time for each step by interrupting the ASP solver, was set to 60 seconds for these experiments.

4.3.3.4 Data and Analysis

The experiments results are given in Table 4.13. In experiments, we observed that the number of testable entities, covering arrays sizes and covering array computation times exponentially has increased with the coverage strength t.

In the computation of standard covering arrays and test case covering arrays, we observed that there is a correlation between the size of U-CIT objects computed and the number of entities. In those studies, we realize the fact that array sizes increase as the number of entities to cover increases. However, in this study, there is no correlation between entity sizes and U-CIT object sizes. For example, when $t = 1$, while 152 entities for parrot model is covered by 10.0 test cases, it is adequate to cover cherokee model with 172 decisions with 5.0 test cases. Similarly, when $t = 2$, while gimp model with 16438 entities can be 48 configuration, for xfig model with 26985 entities, all 2-way decision combinations have been covered with 45.8 test

cases.

In our experiments, we compared our approach with cover-and-generate constructor proposed by Mercan et al. (2020). In most of experiments, both when $t = 1$ and $t = 2$, our approach computed smaller sized U-CIT objects. More specifically, when t=1, U-CIT object sizes are improved up to 9.1%, when t=2, UCIT-ASP computes smaller U-CIT object up to 21.4% in comparison with cover-and-generate constructor.

			ucit-asp		**cover-and-generate**		
t	**model**	**entities**	**size**	**time**	**size**	**time**	**size improvement (%)**
1	mpsolve	30	3.0	0.67	3.0	0.31	0.0
	dia	42	6.0	0.67	4.2	0.34	-42.9
	irissi	70	4.0	35.00	4.0	0.66	0.0
	xterm	78	5.0	61.00	4.2	0.58	-19.0
	parrot	152	10.0	69.33	10.0	1.95	0.0
	gimp	198	8.0	122.00	8.0	2.27	0.0
	pidgin	199	4.0	121.00	4.4	2.29	9.1
	python	210	4.0	120.67	4.4	2.07	9.1
	xfig	237	6.0	121.33	6.0	2.74	0.0
	vim	239	6.0	121.67	6.2	2.69	3.2
	sylpheed	258	7.0	122.33	6.6	3.04	-6.1
	cherokee	272	5.0	121.67	5.0	3.53	0.0
2	mpsolve	296	11.0	4.00	14.0	2.07	21.4
	dia	734	18.0	12.33	19.4	2.26	7.2
	irissi	2102	21.0	742.67	24.2	13.16	13.2
	xterm	2871	20.0	722.67	21.2	5.74	5.7
	parrot	10359	52.0	2679.33	55.8	46.65	6.8
	gimp	16438	44.3	3629.67	48.0	67.11	7.6
	pidgin	17857	32.0	2651.67	33.4	31.82	4.2
	python	21180	32.3	3029.00	34.6	28.68	6.6
	xfig	26985	42.3	4071.67	45.8	78.54	7.6
	vim	27442	46.3	4181.67	48.6	56.47	4.7
	sylpheed	31597	44.3	4574.67	47.4	78.2	6.5
	cherokee	32530	38.7	3698.33	45.0	79.89	14.1

Table 4.13 The results of decision coverage experiments.

4.3.3.5 Discussion

Although the UCIT-ASP method was slower in terms of computation time than the cover-and-generate method developed in the literature (Mercan et al. (2020)), it mostly computed smaller U-CIT objects. Particularly, in experiments when the coverage strength is 2 ($t = 2$), UCIT-ASP method reduced the sizes of all objects generated. This shows that the effectiveness of our approach increases as coverage strength increase in this study.

4.4 Computing Def-Use Pair Covering Arrays as U-CIT Objects

In this section, we introduce a novel coverage criterion which is def-use pair covering arrays as a black-box testing strategy and explain how to model def-use pairs in ASP encoding by examples.

4.4.1 Def-Use Pair Covering Arrays

Def-use-based testing strategies in the literature aim to test def-use pairs on the basis of the program variables as a white-box testing method (Su, Wu, Miao, Pu, He, Chen & Su (2017)). In this study, we propose a def-use-based test strategy which is used as a black-box testing strategy to test def-use pairs for symbolic objects defined on graph-based models.

Although a coverage criterion has been developed to test all t-node orders in the literature (Mercan et al. (2020)), the effectiveness of these methods can be further increased. Namely, in the order-based studies, while it is sufficient to cover t-node orders with any path in a graph, the purpose of our study is to ensure that after an object is updated within the nodes, this update must be tested at every node where the object is used without any its state changes. In other words, if an o object is defined in the v_i node and used in v_j node, it is sufficient to find a path starting from v_i to v_j in the order-based studies. However, in def-use pair testing, we must find a path from v_i to v_j such that, in that path, the object is not updated again in nodes visited between them. Thus, the effects of each update on the objects can be tested. If the object is updated by another node on the path, the effect of the change made on that in v_i may disappear, and therefore this change in v_j may not be tested.

The proposed method in graph-based models used in mobile applications where states stand for application screens and edges stands for transition between edges, by defining common resources (variables, inputs, files, and databases, etc.) as objects on applications screens, interactions between screens defining these resources and screens using those resources can be tested. Similarly, in multi-threaded application models in which states represent atomic blocks and edges represents a transition between these atomic blocks, by defining defined and used resources (variable, inputs, files, and databases, etc.) as an object on models, the interaction between atomic

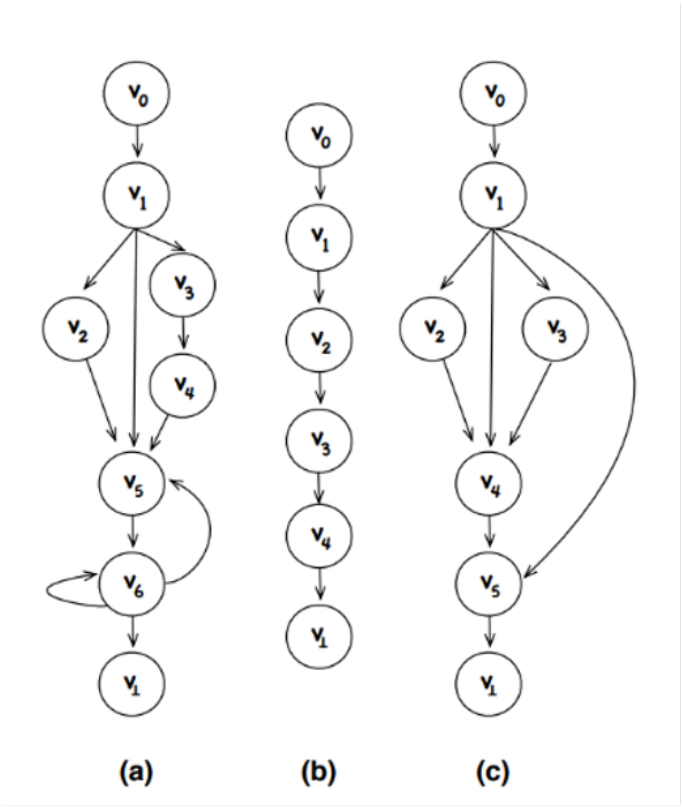


Figure 4.12 An example for graph based models.

blocks using common resources and atomic blocks defining these resources can be tested.

Definition 6 *Given a graph* $G = (V, E, O, D, U, v_0, v_\perp)$ *where* V *is the set of nodes;* E *is the set of edges;* $v_0, v_\perp \in V$ *are the entry and exit nodes respectively;* $O = \{o_1, o_2, \dots\}$ *are symbolic objects defined;* $D = \{D_1, D_2, \dots\}$ *and* $U = \{U_1, U_2, \dots\}$ *refers to nodes where the objects are defined and used respectively. Each set of* $D_i \in D$ *and* $U_i \in U$ *is actually a subset of* V *(*$D_i \subset V$ *and* $U_i \subset V$*). Also, for any object* $o_i \in O$*, the nodes where* o_i *is defined are in the* $D_i \in D$ *set, and the nodes it is used in the* $U_i \in U$ *set.*

Definition 7 *Given* $G = (V, E, O, D, U, v_0, v_\perp)$*, a path is an ordered sequence of nodes* $(v_{i_1}, \dots, v_{i_n})$ *such that* $(v_{i_j}, v_{i_{j+1}}) \in E$ *for* $1 \leq j < n$*.*

Definition 8 *Given* $G = (V, E, O, D, U, v_0, v_\perp)$*, a test case* ϕ *is a path from* v_0 *to* $v_\perp$*. For a given test case* ϕ *with length n,* ϕ_i *where* $1 \leq i \leq n$ *is assumed to be the node having* i^{th} *position in the test case. Hence,* $\phi_1 = v_0$ *and* $\phi_n = v_\perp$*.*

Definition 9 *Given a graph* $G = (V, E, O, D, U, v_0, v_\perp)$*,* $[v_{i_1}, \dots, v_{i_t}]$ *is a t-order* $(v_{i_j} \in V, 1 \leq j \leq t)$*.*

Definition 10 *Given* $G = (V, E, O, D, U, v_0, v_\perp)$*, a tuple of* (v_{i_1}, v_{i_2}, o_j) *is a testable*

def-use entity, if v_{i_1} defines the o_j object ($v_{i_1} \in D_j$), v_{i_2} node uses o_j object ($v_{i_2} \in U_j$), and if there is a path from v_{i_1} to v_{i_2} where o_j object is not redefined (except v_{i_2}).

Definition 11 *Given $G = (V, E, O, D, U, v_0, v_\perp)$, def-use coverage criterion covers all testable def-use entities (in other words, def-use pairs).*

Definition 12 *Given $G = (V, E, O, D, U, v_0, v_\perp)$, a def-use-based U-CIT object is a set of U-CIT test cases that provides full coverage under def-use coverage criterion.*

For example, let $O = \{o_1, o_2\}$ objects are defined for the graph-based model given in Figure 4.12.c, and the definition and usage information of these objects are as follows:

$$
\begin{aligned}
D_1 &= \{v_2\} && \text{\# nodes where } o_1 \text{ is defined} \\
U_1 &= \{v_4, v_5\} && \text{\# nodes where } o_1 \text{ is used} \\
D_2 &= \{v_2, v_4\} && \text{\# nodes where } o_2 \text{ is defined} \\
U_2 &= \{v_4, v_5\} && \text{\# nodes where } o_2 \text{ is used}
\end{aligned}
$$

In this case, def-use coverage criterion marks the following def-use entities to cover:

$$
\begin{aligned}
&(v_2, v_4, o_1) \\
&(v_2, v_5, o_2) \\
&(v_4, v_5, o_2)
\end{aligned}
$$

Note that (v_2, v_5, o_2) is not a valid def-use entity. The reason is that, it is not definitely possible to find a path from v_2 to v_5 without pass through v_4. Since v_4 node defines o_2 object (v_2, v_5, o_2), it is not a valid def-use pair. A def-use U-CIT object to be computed for this scenario will consist of a single test case containing the path $[v_2, v_4, v_5]$.

4.4.2 U-CIT Formulation

In this section, def-use pair covering arrays will be formulated and computed by using the UCIT-ASP approach.

`system_model.ucit`: The difference of the graph models used in def-use pair covering arrays from the usual graph models is that there are two sets containing defined and used objects mapped to nodes in these models. Therefore, a graph model prepared to compute a def-use pair covering array has the characteristics of other graph-based models. This shows that the same or similar modeling approaches can be used in modeling different systems and computing related U-CIT objects.

As discussed in Section 3.1, `state(...)`, `edge(...)`, `start_state(...)` and `end_state(...)` rules are used to define graph-based models. In addition to these rules, objects in the model are defined by using `object(...)` rule and the parameter in this rule expresses the name of the defined object. `def(...)` and `use(...)` rules are used to determine the nodes where an object is defined and used respectively. The first parameter in these rules refers to the name of the node where the definition or the use occurs and the second parameter refers to the name of the object defined or used in that node with respect to the rule used.

Figure 4.13 presents an example `system_model.ucit` file. There are two objects defined in the graph model expressed: `o1` and `o2`. For example, `o1` is defined only in node `v2` and used in node `v4` and `v5`. The definition of def-use pairs for this model is computed by using the `single_path_def_clear_path_def` directive of `graphs` library developed (Section 3.4.2). This directive generates the following ASP rules to determine def-use pairs:

```
def_use_pair(A, B, O) :- def(A, O), use(B, O),
    reaches(A, B), not redefs(A, _, B, O).


redefs(A, X, B, O) :- A != X, X != B, def(A, O), def(X, O),
    use(B, O), reaches(A, X), reaches(X, B).
```

Note that these rules requires a path from `A` to `B` where the object `O` is not redefined to satisfy `def_use_pair(A, B, O)` rule, so that the definition of object `O` in node A and the use in node `B` creates a def-use pair.

If graph theory is wanted to use to compute def-use pairs, the `graphs.graph_theory.def_use_pairs` directive can be used. (Section 3.4.3)

`coverage_criterion.ucit`: As shown in Figure 4.14, the coverage criterion defined for this testing scenario is that all valid def-use pairs must be covered.

`test_space.ucit`: As shown in Figure 4.15, a valid test case for this scenario is a path from the start node to the end node.

```
% Start and end nodes
start_state(v0).
end_state(v6).

% Nodes
state(v0).
state(v1).
state(v2).
state(v3).
state(v4).
state(v5).
state(v6).

% Edges
edge(v0, v1).
edge(v1, v2).
edge(v1, v3).
edge(v1, v4).
edge(v1, v5).
edge(v4, v5).
edge(v5, v6).

% Objects
object(o1).
object(o2).

% Object definitions
def(o1, v2).
def(o2, v2).
def(o2, v4).

% Object uses
use(o1, v4).
use(o1, v5).
use(o2, v4).
use(o2, v5).

% Def-clear-path rule
## graphs.graphs.single_path_def_clear_path_def ##
```

Figure 4.13 An example `system_model.ucit` file.

```
% Entity definition
entity(def_use_pair, A, B, O) :- def_use_pair(A, B, O).
```

Figure 4.14 An example `coverage_criterion.ucit` file.

```
% Test case definition
testcase :- reaches(A, B), start_state(A), end_state(B).
```

Figure 4.15 An example `test_space.ucit` file.

4.4.3 Experiment

In this section, the related experiment information will be provided on computing def-use pair covering arrays as U-CIT objects. In the rest of the section, we will present respectively our experiment setup, evaluation framework, operational framework, data and analysis of experiment results and finally, the discussion of experiment results.

4.4.3.1 Setup

A series of experiment has been carried out to evaluate our proposed approach on the covering array generation for def-use pairs. In these experiments, 42 graphs selected from the graphs obtained from real software (Çalpur (2012)) were used as experiment models.

Experiment graph variations were obtained by using the cross product of the following independent variables for each graph mentioned above:

- the number of objects in the diagram: $\{10, 20\}$
- the number of definitions per objects: $\{5, 10\}$
- the number of uses per objects: $\{5, 10\}$

For this purpose, symbolic objects were created in graphs as many as the total number of objects declared above. Each object is defined in the randomly selected nodes and used in the randomly selected nodes as many as aforementioned in the experiment setup. Thus, 8 different experiment models were prepared for each graph model.

4.4.3.2 Evaluation Framework

In this study, we used the same evaluation metrics introduced at Section 4.1.3.2, except the size improvement metric to evaluate.

4.4.3.3 Operational Framework

In the experiments for this study, we used the same operational framework that is stated in Section 4.1.3.3.

4.4.3.4 Data and Analysis

Table 4.14 summarizes the results obtained from experiments. Due to the number of experiments performed to evaluate the approach, this table presents the average results obtained by grouping the experiment results on the basis of the independent variables aforementioned. In other words, the table presents the aggregated result of experiments with respect to the combination of experiment setup parameters.

In the experiments, we observed that the experiment parameters, which are the number of objects, definitions per object, and uses per object, affect construction times and object sizes. While, when the number of objects in graphs is 10, the average U-CIT object size is 6.5, when this number is doubled to 20, the average U-CIT object size becomes 8.0.

Similarly, particularly for graphs containing a large number of representative objects in the graphs, an increase in the number of defs and uses in graphs leads to an increase as well in the covering array sizes. For example, when objects in graphs are 20 and defs per object is 5, doubling uses per object in graphs from 5 to 10, increases U-CIT object sizes $1.98x$ times.

Moreover, if we keep the number of objects and number of uses per object constant, we can observe the same change in U-CIT object size by doubling the number defs per object.

In experiments, two entity enumeration approaches were used which are graph-theory-based enumeration and ASP-based enumeration. The average entity computation time for both approaches is 0.18s. This indicates that both imperative and declarative programming techniques used in the testable entity enumeration perform

similar performances.

						enumeration time (s)		
objects	**defs per object**	**uses per object**	**stats**	**entities**	**size**	**graph-based**	**ASP-based**	**construction time (s)**
10	5	5	min	45.0	3.0	0.00	0.00	0.00
			avg	55.0	5.4	0.08	0.05	3.23
			max	74.0	11.0	0.33	0.33	7.33
		10	min	80.0	3.0	0.00	0.00	0.00
			avg	113.4	6.8	0.12	0.13	4.52
			max	152.0	18.0	0.67	0.33	16.67
	10	5	min	50.0	3.0	0.00	0.00	0.33
			avg	63.2	6.1	0.04	0.06	3.55
			max	103.0	15.0	0.67	0.33	15.67
		10	min	104.0	3.0	0.00	0.00	0.33
			avg	128.1	7.5	0.07	0.09	5.31
			max	219.0	24.0	0.33	0.67	20.67
20	5	5	min	91.0	3.0	0.00	0.00	0.33
			avg	113.3	6.8	0.08	0.06	4.12
			max	151.0	16.0	0.33	0.33	12.00
		10	min	181.0	3.0	0.00	0.00	0.33
			avg	224.6	8.2	0.07	0.08	4.92
			max	310.0	17.7	0.33	0.67	17.67
	10	5	min	103.0	3.0	0.00	0.00	1.00
			avg	125.8	7.7	0.10	0.06	5.72
			max	191.0	21.0	0.33	0.33	20.33
		10	min	215.0	3.0	0.00	0.00	0.67
			avg	256.0	9.1	0.07	0.07	7.27
			max	347.0	23.7	0.67	0.67	25.00

Table 4.14 The experiment results of def-use pair covering array generation.

4.4.3.5 Discussion

As expected, as the number of objects, the number of definitions per object, and the number of uses per object have been increased, both the dimensions of covering arrays and their computation times have increased. Since graph models enriched with objects are unique models within the scope of this study, the results obtained could not be compared with other approaches.

4.5 Computing Path Aware Covering Arrays

In this section, firstly, we will introduce a novel U-CIT object which is path aware covering arrays and explain how they are formulated as U-CIT objects and computed with UCIT-ASP. In addition, experiments results obtained from the experiments to measure the efficiency and effectiveness of UCIT-ASP in the computation of path aware covering arrays will be provided.

4.5.1 Path Aware Covering Arrays

In this problem, we work with graph-based models where each node contains a set of distinct node parameters; and each parameter can be set to a finite amount of values. In the graph, each set of parameters are uniquely defined for each node and nodes must be visited to activate those parameters.

Such graph-based models can be used as a mobile application testing strategy where mobile applications are represented as a graph-based model in which nodes represent application screen; edges represent a transition between those screens; the parameters defined in nodes refer to input fields on related screens. Moreover, multi-threaded systems can be tested with this testing strategy where nodes represent atomic blocks; edges represent transitions between atomic blocks, and parameters defined at nodes represent shared resources between atomic blocks.

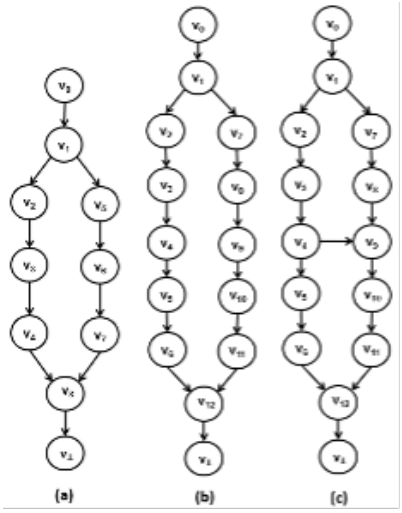


Figure 4.16 An example for graph-based models

When there are parameters defined on the nodes, it might be necessary to test the parameter interactions both within a node and between nodes. For example, the assumption is that there are two screens in a mobile application where round-trip information is entered and all flights are listed with this information. The user wants to make a ticket reservation for an adult and a child over 2 years old on this application. After entering the round-trip information for a certain date, the assumption is that there are enough seats for the selected flight. However, it is not possible to find two seats next to each other. Since it is prohibited that a child over 2 years old to sit in a seat separate from their parents, this case must not be possible. However, if the tests are performed taking into account only the screen transitions (in other words, the order-based studies in the literature (Mercan et al. (2020)) where the interaction between screen inputs is ignored) such errors may not

be detected.

Although standard covering arrays might be a good option to test the interactions of parameters between nodes, most of the test cases to be generated by this method will be invalid.

Therefore, we introduce a new order-based coverage criterion which is based on both the order of nodes and the interactions of the parameters defined in these nodes by expanding the order-based coverage criterion (Mercan et al. (2020)). In the rest of the paper, this coverage criterion is referred as *path aware coverage criteria*.

Definition 13 *Given a graph $G = (V, E, P, K, v_0, v_\perp)$ where; V is the set of nodes, E is the set of edges, $v_0 \in V$ and $v_\perp \in V$ are entry and exit nodes respectively, $P = \{P_1, P_2, \dots\}$ is the parameter sets defined in nodes where $K = \{K_1, K_2, \dots\}$ refers to the finite sets of values that these parameters take. For example, for $v_i \in V$, $P_i = \{p_{i_1}, p_{i_2}, \dots\}$ specifies the parameters defined in this node. The finite set of values that these parameters can take are represented as $K_i = \{k_{i_1}, k_{i_2}, \dots\}$. In other words, the set of $k_{i_j} \in K_i$ refers to the finite set of values $p_{i_j} \in P_i$ can take.*

Definition 14 *For given graph $G = (V, E, P, K, v_0, v_\perp)$, a path is an ordered sequence of nodes $(v_{i_j}, \dots, v_{i_n})$ where a valid value is assigned for all parameters defined by these nodes.*

Definition 15 *For a given $G = (V, E, P, K, v_0, v_\perp)$, a test case θ, is a path starting with node v_0 and ending with node $v_\perp$. Given test case θ of length n, θ_i $(1 \leq i \leq n)$ is assumed to be the node having i^{th} position in the test case. Therefore, $\theta_1 = v_0$ and $\theta_n = v_\perp$.*

Definition 16 *Given a graph $G = (V, E, P, K, v_0, v_\perp)$, $[v_{i_1}, ..., v_{i_t}]$ is a t-way node order ($v_{i_j} \in V$, $1 \leq j \leq t$).*

To cover this node sequence by a test case, nodes must be visited in the order which they are ordered, and a value must be assigned to the parameters within each node from their own set of values. Also, nodes do not have to be visited consecutively. For example, for the graph given in Figure 4.16.a, $(v_0, v_1, v_2, v_5, v_6, v_\perp)$ test case includes both $[v_1, v_2]$ and $[v_1, v_6]$ orderings. On the other hand, $[v_2, v_3]$ is not a valid 2-way order, because there is no path from v_2 to v_3.

Based on these definitions, the coverage criteria given are defined below.

Definition 17 *Path aware coverage criterion with (t', t) parameters, for a given graph $G = (V, E, P, K, v_0, v_\perp)$, marks for each t'-order nodes in G and t-combinations of parameters defined in these nodes to cover.*

Definition 18 *Given graph $G = (V, E, P, K, v_0, v_\perp)$, a path aware U-CIT object is a set of U-CIT test cases that provides full coverage under path aware coverage criterion.*

Notice that the coverage criterion proposed here has two parameters: node order strength (t') and parameter interaction strength (t). While node order strength determines the size of node sequences in interest, the interaction strength determines the coverage strength that must be obtained among parameters in these node orderings.

During this study, 3 ordering criteria has been used which are *any*, *consecutive* and *non-consecutive* node orderings. While these ordering criteria determine in which order nodes must be covered, it also determines the interactions of parameters that must be covered.

In addition to *any* t-order definition described in Definition 16, for consecutive and non-consecutive node orderings introduced in the literature (Mercan et al. (2020)) as well, U-CIT objects will be formulated in Section 4.5.2.

4.5.2 U-CIT Formulation

Path-aware U-CIT objects have been introduced in Section 4.5.1. In this section, these objects will be formulated and computed by using the UCIT-ASP approach. In addition, in this section, a path-aware U-CIT object is defined with only *any* t'-order criterion. In this section, with the flexibility provided by UCIT-ASP, alongside t'-any-order path aware U-CIT object, t'-consecutive-order and t'-nonconsecutive-order path aware U-CIT objects will be formulated in ASP. These U-CIT objects will be evaluated with the experiments carried out with graph models obtained from real software systems (Çalpur (2012)) in the next section.

The models used for path aware U-CIT objects are graph based models. Therefore, `state(...)`, `edge(...)`, `start_state(...)` and `end_state(...)` rules are developed in UCIT-ASP (in Section 3) to define these models. In addition to these rules, `option(...)` rule is defined to express parameters defined in the nodes which can take a finite set of values. For example, the following ASP rule

```
option(s1, o1, 1..4).
```

defines a parameter named `o1` which is defined in node `s1` and can take four different values (1, 2, 3 or 4).

```
start_state(ss).
end_state(se).

state(ss).
state(se).
state(s1).
state(s2).
state(s3).

edge(ss, s1).
edge(s1, s2).
edge(s1, s3).
edge(s2, se).
edge(s3, se).

option(s1, o1, 1..4).
option(s1, o2, 1..2).
option(s2, o3, 1..4).
option(s3, o4, 1..2).

## graphs.graphs.any_order_t1_tuple_t2 ${'t1':2, 't2':2}$ ##
```

Figure 4.17 An example `system_model.ucit` file.

```
% Coverage criterion
entity(any_order, A, O1, V1, B, O2, V2)
    :- any_order_2_tuple_2(A, O1, V1, B, O2, V2).
```

Figure 4.18 An example `coverage_criterion.ucit` file.

```
% Test case
testcase :- reaches(A, B), start_state(A), end_state(B).
```

Figure 4.19 An example `test_space.ucit` file.

`system_model.ucit`: In Figure 4.17, an example `system_model.ucit` file to compute a path aware U-CIT object is presented. To determine the entities to be covered, `any_order_t1_tuple_t2`, `consecutive_order_t1_tuple_t2` and `nonconsecutive_order_t1_tuple_t2` directives have been developed. For a given graph model, these directives, determines the entities to be covered under (t',t) path aware coverage criterion, (t',t) consecutive path aware coverage criterion, and (t',t) non-consecutive path aware coverage criterion respectively. In Figure 4.17, only `any_order_t1_tuple_t2` directive is used as an example, where other directives can be used similarly.

For example, the following directive,

```
## graphs.graphs.any_order_t1_tuple_t2 ${‘t1’:2,s‘t2’:2}$ ##
```

generates the following ASP rules to determine all testable entities that must be covered by $(2,2)$-way any path aware U-CIT objects:

```
reaches(A, A) :- state(A).
any_order_2_tuple_2(A, O1, V1, B, O2, V2) :- A != B,
   option(A, O1, V1), option(B, O2, V2), any_order(A, B).
any_order_2_tuple_2(A, O1, V1, A, O2, V2) :- O1 < O2,
   option(A, O1, V1), option(A, O2, V2).
any_order(A, B) :- reaches(A, B).
```

`any_order_2_tuple_2(A, O1, V1, B, O2, V2)` rule here states that the interaction between the parameter `O1` defined in node `A` value `V1` and the parameter `O2` defined in node `B` with value `V2` must be included for the ordered `A` and `B` node pair. Note that the symbols `A`, `B`, `O1`, `V1`, `O2`, and `V2` in this context are actually variables and these variables will be matched by the ASP solver with all concrete values satisfying the criterion.

Similarly, to determine all the entities to be covered by the $(2,2)$-way consecutive path aware U-CIT object with

```
## graphs.graphs.consecutive_order_t1_tuple_t2 ${‘t1’:2,‘t2’:2}$ ##
```

directive can be used and this directive automatically generates the following ASP rules:

```
edge(S,S) :- state(S).
consecutive_order_2_tuple_2(S1, O1, V1, S2, O2, V2) :- S1 != S2,
   option(S1, O1, V1), option(S2, O2, V2),
   consecutive_order(S1, S2).
```

```
consecutive_order_2_tuple_2(S, O1, V1, S, O2, V2) :- O1 < O2,
   option(S, O1, V1), option(S, O2, V2).
consecutive_order(A, B) :- edge(A, B), A != B.
```

To determine all the entities that must be covered by $(2,2)$-way non-consecutive path-aware U-CIT object with

```
## graphs.graphs.nonconsecutive_order_t1_tuple_t2 ${‘t1’:2,‘t2’:2}$ ##
```

directive can be used and this directive automatically generates the following ASP rules:

```
nonconsecutive_order(A, B)
    :- reaches(A, C), reaches(C, B), A != B, B != C.
nonconsecutive_order_2_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    nonconsecutive_order(S1, S2).
nonconsecutive_order_2_tuple_2(S, O1, V1, S, O2, V2)
    :- O1 < O2,
    option(S, O1, V1), option(S, O2, V2).
nonconsecutive_order(A, B)
    :- reaches(A, C), reaches(C, B), A != B, B != C.
```

More information on how entity definitions are generated for different t' node orderings and t parameter combinations can be found in Appendix A.

`coverage_criterion.ucit`: As can be seen from the sample `coverage_criterion.ucit` file given in Figure 4.18, once the entities are determined, entity rules defining the coverage criterion can be created as follows:

```
entity(any_order, A, O1, V1, B, O2, V2)
    :- any_order_2_tuple_2(A, O1, V1, B, O2, V2).
```

So far, it has been discussed how to determine testable entities with ASP-based formulation. On the other hand, these entities can also be computed with Python scripts by using graph theory. The following ASP directives can be used to compute entities by using graph theoretical approaches:

```
## graphs.graph_theory.path_aware_any_order ${‘t1’,‘t2’}$ ##
## graphs.graph_theory.path_aware_consecutive_order ${‘t1’,‘t2’}$ ##
## graphs.graph_theory.path_aware_nonconsecutive_order ${‘t1’,‘t2’}$ ##
```

`test_space.ucit`. A valid test case for this scenario is a path from the start node to the end node, as shown in Figure 4.19.

4.5.3 Experiment

In this section, the related experiment information will be provided on computing path aware covering arrays as U-CIT objects. In the rest of the section, we will present respectively our experiment setup, evaluation framework, operational framework, data and analysis of experiment results, and finally, the discussion of experiment results.

4.5.3.1 Setup

A series of experiments have been carried out to evaluate the proposed method. In these experiments, 15 graphs obtained from real software were used as graph models (Çalpur (2012)). Graph variations were obtained by using the cross product of the following independent variables for each graph used:

- the number of parameters per node: $\{3, 4\}$
- the number of values per parameters: $\{2, 3\}$
- ordering-interaction strength pairs: $\{(2, 2), (3, 3)\}$

To run the experiments, by computing the cross-product of experiment setup parameters, graphs are populated with parameters and their value sets. As a result, for each graph, 4 different graph models were obtained.

4.5.3.2 Operational Framework

Unless otherwise is stated, all the experiments were repeated 3 times and carried out on Google Cloud machines using Intel Xeon 2.30GHz CPU with 4Gb of RAM, and running 64-bit Ubuntu 18.04 as the operating system. The time limit configuration

parameter of UCIT-ASP, which is used to limit test case generation time for each test case generation step, is set to 45 seconds for these experiments.

4.5.3.3 Evaluation Framework

Except for the size improvement metric, all evaluation metrics introduced in Section 4.1.3.2 have been used to evaluate the proposed approach in the experiments.

4.5.3.4 Data and Analysis

Table 4.15 and Table 4.16 summarize the results obtained from the experiments. Due to the size of the experiment space used, this table has been presented by dividing experiments into four groups by determining each group based on the number of entities in experiments (in a way that we gathered experiments with a similar number of entities in the same group) and taking averages of each group statistics for specific experiment setup aforementioned in Section 4.5.3.1.

In these experiments, we observed that when experiment parameters such as coverage strength, the number of parameters per node and the number of values each parameter defined can take, are increased, the number of testable entities and construction times increase dramatically. To better see the impact of this change, let's consider model group 4, and review entities, U-CIT object sizes, and time measurements. For example, let's consider any-ordered path aware U-CIT objects in this group since experiments in complex graphs show the differences better as they scale. Increasing coverage strength from $(2,2)$ to $(3,3)$ lead to $48x$ times increase on the average entity sizes. Also, average U-CIT object sizes and U-CIT object computation time respectively have increased $4.38x$ and $12.46x$ times for this model group and set of experiment configuration.

4.5.3.5 Discussion

					enumeration time (s)			
model group	**order**	**params**	**values**	**entities**	**graph based**	**ASP based**	**size**	**construction time (s)**
1	any	3	2	2520.0	0.00	0.17	12.00	416.83
		3	3	5670.0	0.42	0.42	24.84	962.09
		4	2	4512.0	0.17	0.33	13.00	501.67
		4	3	10152.0	0.42	0.67	26.58	1067.50
	consecutive	3	2	576.0	0.00	0.17	8.00	190.50
		3	3	1296.0	0.08	0.08	24.00	563.42
		4	2	1056.0	0.25	0.00	10.00	274.67
		4	3	2376.0	0.42	0.33	26.00	734.17
	nonconsecutive	3	2	2124.0	0.08	0.25	11.00	458.50
		3	3	4779.0	0.58	0.17	21.83	1142.00
		4	2	3808.0	0.42	0.50	12.00	547.33
		4	3	8568.0	1.00	0.67	25.58	1297.58
2	any	3	2	4512.0	0.44	0.50	12.00	502.56
		3	3	10152.0	0.56	0.67	26.55	1068.61
		4	2	8064.0	0.67	0.50	14.00	553.17
		4	3	18144.0	1.22	1.17	28.39	1198.11
	consecutive	3	2	840.0	0.22	0.22	12.00	274.50
		3	3	1890.0	0.39	0.11	33.00	783.17
		4	2	1536.0	0.39	0.00	14.11	321.22
		4	3	3456.0	0.56	0.17	34.00	973.61
	nonconsecutive	3	2	3972.0	0.45	0.22	12.00	548.55
		3	3	8937.0	0.83	0.61	25.00	1357.72
		4	2	7104.0	0.78	0.61	12.06	613.56
		4	3	15984.0	1.22	1.06	27.28	1510.56
3	any	3	2	5724.0	0.50	0.34	14.00	505.84
		3	3	12879.0	0.50	0.84	27.84	1123.67
		4	2	10224.0	0.67	1.00	14.50	554.17
		4	3	23004.0	1.17	1.67	29.17	1253.67
	consecutive	3	2	900.0	0.17	0.00	13.00	272.50
		3	3	2025.0	0.17	0.50	33.00	646.50
		4	2	1648.0	0.17	0.17	14.00	321.00
		4	3	3708.0	0.50	0.34	37.67	798.17
	nonconsecutive	3	2	5112.0	0.17	0.33	12.00	552.00
		3	3	11502.0	1.00	0.84	26.50	1416.83
		4	2	9136.0	0.50	0.50	13.00	676.00
		4	3	20556.0	1.84	1.17	28.34	1586.84
4	any	3	2	3996.0	0.00	0.44	13.33	485.11
		3	3	8991.0	0.67	0.33	26.11	1036.89
		4	2	7144.0	0.67	0.56	14.00	524.78
		4	3	16074.0	1.00	0.67	28.11	1168.89
	consecutive	3	2	732.0	0.11	0.11	10.67	247.56
		3	3	1647.0	0.11	0.33	28.33	628.56
		4	2	1341.3	0.11	0.33	11.67	286.00
		4	3	3018.0	0.33	0.33	30.33	734.67
	nonconsecutive	3	2	3492.0	0.11	0.22	11.67	523.11
		3	3	7857.0	0.78	0.33	25.45	1304.11
		4	2	6248.0	0.89	0.44	12.33	616.78
		4	3	14058.0	1.11	1.11	26.56	1472.45

Table 4.15 Information about the $(2,2)$-way path aware U-CIT objects computed in the experiments.

					enumeration time (s)			
model group	**order**	**params**	**values**	**entities**	**graph theoretic**	**ASP based**	**size**	**construction time (s)**
1	any	3	2	57120.0	6.75	4.92	32.33	1475.25
		3	3	192780.0	26.25	19.34	117.00	7411.67
		4	2	138368.0	19.25	12.92	36.00	1841.58
		4	3	466992.0	72.75	51.84	136.00	13337.83
	consecutive	3	2	6144.0	1.00	0.50	34.00	732.08
		3	3	20736.0	3.33	1.92	125.84	4697.75
		4	2	15744.0	2.50	1.67	40.59	1101.17
		4	3	53136.0	8.84	5.67	147.92	7151.92
	nonconsecutive	3	2	54960.0	8.08	5.25	31.84	1906.83
		3	3	185490.0	30.25	20.25	116.67	9756.83
		4	2	133248.0	22.92	14.00	36.17	2469.25
		4	3	449712.0	87.00	54.83	134.42	17473.33
2	any	3	2	138368.0	18.11	14.17	36.39	2009.72
		3	3	466992.0	69.61	52.22	135.06	13271.17
		4	2	333312.0	51.17	35.56	41.28	3054.22
		4	3	1124928.0	190.44	138.11	157.72	26166.22
	consecutive	3	2	9632.0	1.72	1.00	50.00	1289.55
		3	3	32508.0	5.61	3.44	174.33	7035.89
		4	2	24576.0	4.72	2.39	57.17	1919.50
		4	3	82944.0	16.17	9.67	208.11	10992.22
	nonconsecutive	3	2	135344.0	21.94	15.39	36.06	2605.28
		3	3	456786.0	81.84	58.11	134.45	17138.45
		4	2	326144.0	57.89	38.89	40.61	3986.78
		4	3	1100736.0	228.45	152.61	154.94	34894.17
3	any	3	2	198432.0	27.00	20.34	38.17	2581.17
		3	3	669708.0	103.00	78.16	142.83	17306.34
		4	2	477120.0	75.00	50.50	42.83	3232.34
		4	3	1610280.0	281.34	210.00	202.33	32969.34
	consecutive	3	2	10080.0	1.50	0.84	53.00	988.00
		3	3	34020.0	6.17	3.50	188.33	7511.67
		4	2	25792.0	4.17	2.84	62.34	1875.50
		4	3	87048.0	15.34	9.67	221.33	11302.83
	nonconsecutive	3	2	194976.0	32.67	22.17	37.67	3337.17
		3	3	658044.0	123.34	85.17	143.17	22832.50
		4	2	468928.0	89.34	58.00	42.17	4435.67
		4	3	1582632.0	342.00	226.00	165.83	41430.50
4	any	3	2	116760.0	14.89	11.67	35.11	1823.55
		3	3	394065.0	59.44	42.56	130.55	11366.78
		4	2	281482.7	42.22	30.56	39.67	2810.33
		4	3	950004.0	161.78	114.22	152.66	24076.78
	consecutive	3	2	7872.0	1.44	0.78	39.78	789.89
		3	3	26568.0	4.44	2.45	143.33	5513.67
		4	2	20149.3	3.56	1.89	46.33	1292.67
		4	3	68004.0	12.89	7.78	168.45	8691.00
	nonconsecutive	3	2	113952.0	17.78	12.33	35.00	2389.33
		3	3	384588.0	66.78	45.67	130.00	15013.00
		4	2	274826.7	50.33	32.11	39.44	3630.33
		4	3	927540.0	187.89	126.00	149.56	30944.00

Table 4.16 Information about the (3,3)-way path aware U-CIT objects computed in the experiments.

As expected, when the number of parameters per node, the number of values per parameter, and/or the coverage strengths are increased, both the size of path aware U-CIT objects and the computation times of these objects and the number of testable entities increased.

In addition, as expected, the number of testable entities in the same experimental setups is generally ordered from smallest to largest for consecutive ordered, non-consecutive ordered, and any ordered coverage criterion. Although this situation caused the sizes of computed path aware U-CIT objects to decrease in the same order, exceptions were also observed due to the randomness in the computation methods of these objects and the reachability constraints arising from the graph models used. When the graph-theory-based and ASP-based approaches are compared in terms of computing testable entities under a given coverage criterion, it is observed that the ASP-based method computes the testable entities in equal or less time than the graph-theory-based approach. This shows that the ASP formulation that we developed is efficient.

5. THREATS TO VALIDITY

Internal and external threats to the validity of all empirical studies exist. In our study, we are concerned about both external and internal validity threats, because they restrict our approach to generalize the result of our experiments to industrial practice.

5.1 Internal Validity

The system that we developed to compute U-CIT objects have many configuration parameters, which are `coverage_max_entities`, `max_entities` and `time_limit`. We set these parameters based on our observation in experiments to minimize both computation times and object sizes. However, these configuration values presumably are not set to the global optimum values of configuration parameters. Hence, we cannot guarantee that experiment results present covering arrays with the smallest sizes which can be computed with this approach.

5.2 External Validity

Within the scope of this study, we evaluated our approach proposed by caring out five different case studies in which different CIT (both already existing in the literature and newly introduced) objects are generated by using UCIT-ASP. In these evaluations, one of the possible threats against our approach is the representativeness of objects generated in CIT area. We generated well-known three CIT objects in CIT, which are standard covering arrays, test case aware covering arrays, and

decision covering arrays. However, since we didn't compute all CIT objects defined in the literature, the representativeness of this research can be only measured on these case studies.

Secondly, in experiments, we tried to test as much as possible models obtained from real software systems. However, the models might not be representative for testing all kinds of software systems. Since it is not possible to test our approach with every software system that exists, we tried to experiment with experiment models obtained from well-known software systems used in the industry.

6. RELATED WORK

The computation of traditional standard covering arrays is an NP-complete problem. Nie and Leung state that approximately 50 papers were published in the last 20 years and their main theme is only to compute standard covering arrays (Nie & Leung (2011)). This number emphasizes the importance of efficiency, effectiveness, and scalability of methods used in CIT object computation, specifically in traditional covering array generation, for both academic and practical purposes.

Traditional covering array generation methods can be examined 4 broad categories: opportunistic (greedy) methods (Cohen et al. (1997); Lei et al. (2007); Wang & He (2013); Wagner, Kleine, Simos, Kuhn & Kacker (2020)), mathematical methods (Kobayashi (2002); Colbourn (2004)), methods based on random search (Schroeder, Bolaki & Gopu (2004); Huang, Xie, Chen & Lu (2012)), and meta-heuristic methods (Bryce & Colbourn (2007); Wu, Nie, Kuo, Leung & Colbourn (2015); Galinier, Kpodjedo & Antoniol (2017); Jia et al. (2015); Torres-Jimenez & Rodriguez-Tello (2012))

Opportunistic methods work iteratively (Cohen et al. (1997)). At each iteration, an uncovered t-way parameter value combination from a set of configurations evaluated as candidates that covering the most possible configurations is added to the covering array and marked as covered in all t-way combinations covered by that configuration. The iterations end when all t-way combinations are covered by the selected configurations.

Mathematical methods have also been developed to compute traditional standard covering arrays (Kobayashi (2002); Hartman (2005); Ji & Yin (2010); Colbourn (2014)). Generally, while these methods compute standard covering arrays for large configuration space models (i.e. models with a large number of configuration parameters), they recursively use covering arrays computed for small parts of these models (Kobayashi (2002)).

Random search-based methods use a random search without replacement strategy (Chen, Kuo, Merkel & Tse (2010)). In this method, a configuration is randomly

selected from the current configuration space model at each iteration. Iterations continue until the selected set of configurations forms a t-way covering array. This method is often used where other available methods do not scale.

Meta-heuristic methods, on the other hand, use either search-based techniques such as hill climbing (Cohen, Colbourn & Ling (2003)), tabu-search (Bryce & Colbourn (2007)), and simulated annealing (Cohen et al. (2003); Lin, Luo, Cai, Su, Hao & Zhang (2015); Rodriguez-Cristerna, Torres-Jimenez, Gómez & Pereira (2015)) or artificial intelligence based techniques such as genetic algorithms (Bansal, Mittal, Sabharwal & Koul (2014)) and ant colony (Ahmed, Zamli & Lim (2012)) to compute traditional covering arrays. These methods always have a set of candidate configurations, which are iteratively transformed until a traditional t-way covering array forms.

All these methods have been developed to compute traditional covering arrays. In this paper, we have developed unique methods to compute both traditional covering arrays (since these CIT objects are a special case of U-CIT objects) and introduced novel U-CIT objects (Section 4.4 and 4.5). In theory, the above-mentioned methods have the potential to be adapted to compute U-CIT objects. However, particularly, we have adapted opportunistic methods in our studies due to their flexibility in practice.

In real life, all possible combinations of parameter values in a configuration space model may not be valid (Calvagna & Gargantini (2010); Garvin, Cohen & Dwyer (2011); Rao, Li, Lei, Kacker, Kuhn & Guo (2019)). In such cases, marking invalid combinations is achieved by the constraints added to the configuration space model and defined over parameter values. For example, a constraint that can be used in a system defining the operating system (OS) and Web browser (WEB) to be used as two separate configuration parameters might be that MS Internet Explorer cannot be used with Linux-based operating systems (OS = Linux $\implies$ WEB $\neq$ Internet Explorer). The reason is that Internet Explorer does not support non-Windows-based operating systems. These constraints are called system constraints in the rest of the document.

Cohen et al. have shown that computing covering arrays without complying with the system constraints might cause cost increases due to invalid configuration combinations (Cohen, Dwyer & Shi (2007)). On the other hand, Yılmaz et al. show that ignored constraints cause masking effects and prevent testing of all parameter combinations (Yilmaz, Dumlu, Cohen & Porter (2014)). Grindal et al. propose various methods to satisfy system constraints (Mats, Jeff & Jonas (2006)). Instead of hard constraints, Bryce et al. define soft constraints that allow marking combinations

that are valid but not preferred to be covered more than once (Bryce & Colbourn (2006)). Yılmaz shows that besides the system constraints, there are also test case-specific constraints, and he develops a unique covering array, called test case aware covering array (Yilmaz (2013)). CIT methods have been used many times in testing software systems containing a large number of constraints (Johansen et al. (2012); Henard, Papadakis, Perrouin, Klein & Traon (2013); Devroey, Perrouin, Legay, Cordy, Schobbens & Heymans (2014); Duan, Lei, Kacker & Kuhn (2019)).

In U-CIT, each combination (in general terms, each testable entity) to be covered is expressed as a constraint (Mercan et al. (2020)). Therefore, UCIT-ASP also widely uses constraints between parameters (Section 3.1). However, as discussed in detail in this paper, while U-CIT constraints are applied on the basis of each selected configuration (each configuration must satisfy only a subset of constraints that can be solved together, not all constraints), the aforementioned constraints are applied to the covering array (all configurations in the covering array must satisfy all system-wide constraints).

In covering arrays, it does not matter in what order values are assigned to each configuration parameter in a configuration, as long as it has a valid value. In other words, it is assumed that the system under test will behave in the same way regardless of the order as the configuration parameters are configured. However, in event-based systems, the behavior of the system usually depends on the event occurrence order. Therefore, it is possible to detect different errors by changing the order of occurrences of events. To test such input spaces, CIT objects have been proposed under the name of order-based covering arrays where the order of events is important (Kuhn, Higdon, Lawrence, Kacker & Lei (2012)). An order-based t-way covering array computed for events in a given system is an ordered set of events constructed to include all valid event queues of length t at least once (consecutively or not). Thus, order-based covering arrays allow to efficiently and effectively detect errors occurred by the order of occurrence of t or fewer events. As with traditional covering arrays, path aware covering arrays, which extend order-based coverage criterion, are a special case for U-CIT objects proposed in this paper (Section 4.5).

In cases where the known CIT objects are insufficient, the definition of new coverage criterion and new CIT objects continues without slowing down (Yuan et al. (2011); Yang, Yan & Rountev (2013); Yang, Yan, Wu, Wang & Rountev (2015); Terragni & Cheung (2016); Choudhary, Lu & Pradel (2017)). The one of the main reasons of this is that CIT methods can be used in many areas. For example; Terragni/Choudhary et al., for testing multi-threaded systems (Terragni & Cheung (2016); Choudhary et al. (2017)); Yuan et al. to test graphical interfaces (Yuan et al. (2011)) Colbourn

et al. to find not only errors but also possible combinations of parameter values that cause errors (Colbourn & Fan (2016)); Ronneseth et al., to combine traditional and order based covering arrays (Ronneseth & Colbourn (2009)); Yang et al., on the other hand, to find the causes of the "program not responding" problem due to complex operations in Android programs, propose new CIT-based coverage criteria (Yang et al. (2013); Yang et al. (2015)).

The fact that CIT methods are used in many fields shows the practical and academic value added from the generalization of these methods, and the definitions of these objects declaratively are the main purpose of the project.

7. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a new approach referred as UCIT-ASP to compute U-CIT objects by using a declarative modeling language named as ASP and implemented a novel U-CIT constructor named as one-test-case-at-a-time constructor to compute U-CIT objects within the scope of this study. The flexibility of modeling test scenarios declaratively has allowed us to define novel coverage criteria for different types of systems. To represent systems having different typologies, we also have developed modeling libraries in UCIT-ASP. Specifically, we have provided modeling libraries in which ASP directives generate ASP code to define coverage criterion definitions.

By using these libraries, we carried out 5 different case studies to compute different U-CIT objects and in these studies we evaluated our approach (proposed in Section 3). Although the motivation of the approach to bring easiness and flexibility to define novel U-CIT objects under under certain coverage criteria, we have observed improvements in the covering array sizes generated for already defined CIT objects. In other words, while we provide a tool to declaratively define coverage criteria, also our approach shows an improvement in the covering array sizes generated in these studies.

We consider that as future work many novel coverage criteria can be introduced by using UCIT-ASP approach. We developed modeling libraries to model graph-based models and configuration systems. These libraries can be also extended and new libraries can be developed for other types of software systems as future work. The coverage criteria for graph-based models that we developed in this paper are important to test the interaction between graph nodes for graph-based systems. Therefore, using these U-CIT objects on the testing of graph-based real-life applications such as mobile applications and multi-threaded systems can be an interesting study to see the efficiency of these coverage criteria that we introduced.

ASP provides a certain level of abstraction on modeling of coverage criteria and valid test spaces. However, even a higher level of a modeling language can be provided

as a front-end on top of it. For example, as future work, a domain-specific language can be developed to increase the usability of this approach further.

UCIT-ASP have many configuration parameters such as `time_limit`, `max_entities`, `coverage_max_entities`, which are tuned before generating U-CIT objects. In our studies, we used the most optimal configuration setup by manually tuning the configuration parameters. However, the values of these configuration parameters dramatically change across different scenarios. Therefore, an internal hyper-parameter optimization module can be developed to fine-tune these parameters.

BIBLIOGRAPHY

Ahmed, B. S., Zamli, K. Z., & Lim, C. P. (2012). Application of particle swarm optimization to uniform and variable strength covering array construction. *Applied Soft Computing*, *12*(4), 1330–1347.

Bansal, P., Mittal, N., Sabharwal, A., & Koul, S. (2014). Integrating greedy based approach with genetic algorithm to generate mixed covering arrays for pair-wise testing. *2014 Seventh International Conference on Contemporary Computing (IC3)*.

Bryce, R. C. & Colbourn, C. J. (2006). Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, *48*(10), 960 – 970. Advances in Model-based Testing.

Bryce, R. C. & Colbourn, C. J. (2007). One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, (pp. 1082–1089)., New York, NY, USA. ACM.

Çalpur, M. Ç. (2012). Interleaving coverage criteria oriented testing of multithreaded applications. Master's thesis.

Calvagna, A. & Gargantini, A. (2010). A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, *45*(4), 331–358.

Chen, T. Y., Kuo, F.-C., Merkel, R. G., & Tse, T. (2010). Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, *83*(1), 60–66.

Choudhary, A., Lu, S., & Pradel, M. (2017). Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.

Cohen, D., Dalal, S., Fredman, M., & Patton, G. (1997). The aetg system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, *23*(7), 437–444.

Cohen, M. B., Colbourn, C. J., & Ling, A. C. H. (2003). Augmenting simulated annealing to build interaction test suites. In *Proc. of the 14th Int'l Symposium on Software Reliability Engineering*, (pp. 394–405).

Cohen, M. B., Dwyer, M. B., & Shi, J. (2007). Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, (pp. 129–139).

Colbourn, C. (2014). Conditional expectation algorithms for covering arrays. *The journal of combinatorial mathematics and combinatorial computing*, *90*, 97–115.

Colbourn, C. J. (2004). Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, *58*(121-167), 0–10.

Colbourn, C. J. & Fan, B. (2016). Locating one pairwise interaction: Three recursive constructions. *Journal of Algebra Combinatorics Discrete Structures and Applications*, *3*(3).

Demiroz, G. & Yilmaz, C. (2012). Cost-aware combinatorial interaction testing. In *Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles*, (pp. 9–16).

Devroey, X., Perrouin, G., Legay, A., Cordy, M., Schobbens, P.-Y., & Heymans, P. (2014). Coverage criteria for behavioural testing of software product lines.

Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change Lecture Notes in Computer Science, 336–350.

Duan, F., Lei, Y., Kacker, R. N., & Kuhn, D. R. (2019). An approach to t-way test sequence generation with constraints. *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

Galinier, P., Kpodjedo, S., & Antoniol, G. (2017). A penalty-based Tabu search for constrained covering arrays. In *Proceedings of the Genetic and Evolutionary Computation Conference*, (pp. 1288–1294). ACM.

Garvin, B. J., Cohen, M. B., & Dwyer, M. B. (2011). Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, *16*(1), 61–102.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *6*(3), 1–238.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2018). Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, *19*(1), 27–82.

Gebser, M., Kaufmann, B., & Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, *187-188*, 52–89.

Hartman, A. (2005). Software and hardware testing using combinatorial covering suites. In Golumbic, M. C. & Hartman, I. B.-A. (Eds.), *Graph Theory, Combinatorics and Algorithms*, volume 34, (pp. 237–266). Springer US.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., & Traon, Y. L. (2013). Multi-objective test generation for software product lines. *Proceedings of the 17th International Software Product Line Conference on - SPLC 13*.

Huang, R., Xie, X., Chen, T. Y., & Lu, Y. (2012). Adaptive random test case generation for combinatorial testing. In *Proceedings of 36th IEEE Annual International Computer Software and Applications Conference, (COMPSAC)*, (pp. 52–61). IEEE.

Javeed, A. (2015). Gray-box combinatorial interaction testing. Master's thesis.

Javeed, A. & Yilmaz, C. (2015). Combinatorial interaction testing of tangled configuration options. *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

Jenkins, B. (2005). jenny: A pairwise testing tool. `http://www.burtleburtle.net/bob/index.html`.

Ji, L. & Yin, J. (2010). Constructions of new orthogonal arrays and covering arrays of strength three. *Journal of Combinatorial Theory, Series A*, *117*(3), 236–247.

Jia, Y., Cohen, M. B., Harman, M., & Petke, J. (2015). Learning combinatorial interaction test generation strategies using hyperheuristic search. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

Johansen, M. F., Haugen, Ø., & Fleurey, F. (2012). An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, (pp. 46–55). ACM.

Kobayashi, N. (2002). *Design and evaluation of automatic test generation strategies for functional testing of software*. PhD thesis, Osaka University, Osaka, Japan.

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

Lei, Y., Carver, R. H., Kacker, R., & Kung, D. (2007). A combinatorial testing strategy for concurrent programs. *Software Testing, Verification and Reliability*, *17*(4), 207–225.

Lin, F. & Zhao, Y. (2004). Assat: computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, *157*(1-2), 115–137.

Lin, J., Luo, C., Cai, S., Su, K., Hao, D., & Zhang, L. (2015). Tca: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, (pp. 494–505). IEEE.

Mats, G., Jeff, O., & Jonas, M. (2006). Handling constraints in the input space when using combination strategies for software testing. Technical Report HS-IKI -TR-06-001, University of Skövde, School of Humanities and Informatics.

Mercan, H., Javeed, A., & Yilmaz, C. (2020). Flexible combinatorial interaction testing. *accepted for publication in IEEE Transactions on Software Engineering, doi=10.1109/TSE.2020.3010317*.

Nie, C. & Leung, H. (2011). A survey of combinatorial testing. *ACM Computing Surveys*, *43*(2), 1–29.

Rao, C., Li, N., Lei, Y., Kacker, R. N., Kuhn, D. R., & Guo, J. (2019). Using parameter mapping to avoid forbidden tuples in a covering array. *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

Rodriguez-Cristerna, A., Torres-Jimenez, J., Gómez, W., & Pereira, W. (2015). Construction of mixed covering arrays using a combination of simulated annealing and variable neighborhood search. *Electronic Notes in Discrete Mathematics*, *47*, 109–116.

Ronneseth, A. H. & Colbourn, C. J. (2009). Merging covering arrays and compressing multiple sequence alignments. *Discrete Applied Mathematics*, *157*(9), 2177–2190.

Schroeder, P. J., Bolaki, P., & Gopu, V. (2004). Comparing the fault detection effectiveness of n-way and random test suites. In *Proc. of the 2004 Int'l Symp. on Empirical Software Engineering*, (pp. 49–59).

Schroeder, P. J., Faherty, P., & Korel, B. (2002). Generating expected results for automated black-box testing. In *In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE 2002*, (pp. 139–148). IEEE.

Su, T., Wu, K., Miao, W., Pu, G., He, J., Chen, Y., & Su, Z. (2017). A survey on data-flow testing. *ACM Computing Surveys*, *50*(1), 1–35.

Terragni, V. & Cheung, S.-C. (2016). Coverage-driven test code generation for concurrent classes. *Proceedings of the 38th International Conference on Software Engineering*.

Torres-Jimenez, J. & Rodriguez-Tello, E. (2012). New bounds for binary covering arrays using simulated annealing. *Information Sciences*, *185*(1), 137–152.

Wagner, M., Kleine, K., Simos, D. E., Kuhn, R., & Kacker, R. (2020). Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays. *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

Wang, Z. & He, H. (2013). Generating variable strength covering array for combinatorial software testing with greedy strategy. *JSW*, *8*(12), 3173–3181.

Williams, A. W. & Probert, R. L. (1996). A practical strategy for testing pair-wise coverage of network interfaces. In *Proceedings of Seventh International Symposium on Software Reliability Engineering*, (pp. 246–254). IEEE.

Wu, H., Nie, C., Kuo, F.-C., Leung, H., & Colbourn, C. J. (2015). A discrete particle swarm optimization for covering array generation. *IEEE Transactions on Evolutionary Computation*, *19*(4), 575–591.

Yamada, A., Biere, A., Artho, C., Kitamura, T., & Choi, E.-H. (2016). Greedy combinatorial test case generation using unsatisfiable cores. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*.

Yang, S., Yan, D., & Rountev, A. (2013). Testing for poor responsiveness in android applications. *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*.

Yang, S., Yan, D., Wu, H., Wang, Y., & Rountev, A. (2015). Static control-flow analysis of user-driven callbacks in android applications. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

Yilmaz, C. (2013). Test case-aware combinatorial interaction testing. *IEEE Transactions on Software Engineering*, *39*(5), 684–706.

Yilmaz, C., Cohen, M. B., & Porter, A. (2004). Covering arrays for efficient fault characterization in complex configuration spaces. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA 04*.

Yilmaz, C., Dumlu, E., Cohen, M. B., & Porter, A. (2014). Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. *IEEE Transactions on Software Engineering*, *40*(1), 43–66.

Yilmaz, C., Fouche, S., Cohen, M. B., Porter, A., Demiroz, G., & Koc, U. (2014). Moving forward with combinatorial interaction testing. *Computer*, *47*(2), 37–45.

Yu, L., Lei, Y., Kacker, R. N., & Kuhn, D. R. (2013). Acts: A combinatorial test generation tool. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*.

Yu, Y. T. & Lau, M. F. (2006). A comparison of mc/dc, mumcut and several other coverage criteria for logical decisions. *Journal of Systems and Software*, *79*(5), 577–590.

Yuan, X., Cohen, M. B., & Memon, A. M. (2011). GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, *37*(4), 559–574.

APPENDIX A

ASP Rules Generated by Directives in UCIT-ASP

In this section, we provide ASP rules generated by directives in UCIT-ASP libraries within in the scope of this work. Parametrically generated ASP rules are explained by examples. These directives are adapted for different parameter values and their functionality can be extended for other value combinations.

Ucit Library

This modeling library contains ASP directives to model decision systems.

`decision_system ${‘input_mode’,[‘file’|‘decisions’]}$`

This directive takes a decision system as an input and generates ASP rules for the logical expressions for this decision system. This directive takes two different inputs as parameters, `import` and `process`. If the decision system is to be provided as a file, the `file` argument is provided with the file name as the second argument. If decisions are to be written into the directive, the decision system is inserted into `decision` argument made with this argument as the second argument. In both cases, decisions are expressed in JSON format.

For example, following ASP rules are generated, for JSON file given in Figure A.1:

```
decision1 :- ucit_condition1.
decision1 :- ucit_condition2.
1{var(ucit_a, 1); var(ucit_a, 2)} 1.
1{var(ucit_b, 1); var(ucit_b, 2)} 1.
ucit_condition1 :- var(ucit_a,UCIT_a), UCIT_a == 1.
ucit_condition2 :- var(ucit_b,UCIT_b), UCIT_b == 2.
```

```
{
  "variables":[{
         "name":"a",
         "values":[ 1, 2 ]
      },{
         "name":"b",
         "values":[ 1, 2 ]
      }],
  "decisions":[{
         "id":"decision1",
         "decision":"(a == 1 and b == 2)"
      }]
}
```

Figure A.1 An example decision file provided in JSON format.

bool_var ${‘vars’}$

This directive generates ASP rules for each Boolean variable in `vars` variable list.

For example, for the directive below,

```
## bool_var  ${‘vars’: [‘a’, ‘b’]}$ ##
```

following ASP rules are generated:

```
parameter(global, a, true).
parameter(global, a, false).
parameter(global, b, true).
parameter(global, b, false).
```

bool_expr ${‘name’,‘expr’} $

Generates ASP rules for a given Boolean expression. The variables that generates Boolean expression must have been previously defined using the `bool_var` directive.

For example, for the directive below,

```
## bool_expr  ${‘name’: ‘decision1’, ‘expr’: ‘(a & b)’}$ ##
```

the following ASP rule is generated:

```
decision1 :- parameter(global, a, true), parameter(global, b, true).
```

Configs Library

This modeling library contains ASP directives to model configuration systems.

`minimal_forbidden_tuple ${‘tuple’}$`

Generates ASP code to defines minimal forbidden parameter value combinations. For example, for $o1 = 1$ and $o2 = 2$ parameter combination, following ASP code is generated:

```
minimal_forbidden_tuple(o1, 1, o2, 2).
```

`t_tuple ${‘t’}$`

Generates ASP code to express t-tuple parameter value combinations. For example, this directive generates the following code,

when $t = 2$,

```
t_tuple(O1, V1, O2, V2)
    :- O1 < O2,
    option(O1, V1), option(O2, V2),
    not minimal_forbidden_tuple(O1, V1),
    not minimal_forbidden_tuple(O2, V2),
    not minimal_forbidden_tuple(O1, V1, O2, V2).
```

and when $t = 3$,

```
t_tuple(O1, V1, O2, V2, O3, V3)
    :- O1 < O2, O2 < O3,
    option(O1, V1), option(O2, V2), option(O3, V3),
    not minimal_forbidden_tuple(O1, V1),
    not minimal_forbidden_tuple(O2, V2),
    not minimal_forbidden_tuple(O3, V3),
    not minimal_forbidden_tuple(O1, V1, O2, V2),
    not minimal_forbidden_tuple(O1, V1, O3, V3),
    not minimal_forbidden_tuple(O2, V2, O3, V3),
```

```
not minimal_forbidden_tuple(O1, V1, O2, V2, O3, V3).
```

Graphs Library

This modeling library contains ASP directives to model graph-based systems.

`any_order ${‘t’}$`

Generates ASP code to determine any ordered t nodes. For example, this directive generates the following code,

when $t = 2$,

```
any_order(A, B) :- reaches(A, B).
```

and when $t = 3$,

```
any_order(A, B, C) :- reaches(A, B), reaches(B, C).
```

`consecutive_order ${‘t’}$`

Generates ASP code to determine all consecutive ordered t nodes. For example, this directive generates the following code,

when $t = 2$,

```
consecutive_order(A, B) :- edge(A, B), A != B.
```

and when $t = 3$,

```
consecutive_order(A, B, C) :- edge(A, B), edge(B, C), A != B, B != C.
```

nonconsecutive_order ${'t'}$

Generates ASP code to determine all non-consecutive ordered t nodes. For example following ASP code are generated,

when $t = 2$,

```
nonconsecutive_order(A, B)
    :- reaches(A, C), reaches(C, B), A != B, B != C.
```

and when $t = 3$,

```
nonconsecutive_order(A, B, C)
    :- reaches(A, B), reaches(B, X), reaches(X, C),
    A != B, B != C, C != X.
nonconsecutive_order(A, B, C)
    :- reaches(A, X), reaches(X, B), reaches(B, C),
    A != B, B != C, B != X.
```

def_use_pair_def

Generates ASP code to determine all *def-use pairs* in a given graph. This directive generates following ASP code:

```
def_use_pair(S1, S2, O)
    :-  S1 != S2,
    def(S1, O), use(S2, O), reaches(S1, S2).
```

single_def_clear_path_def

Generates ASP code to determine *def-clear-paths* in a given path. This directive generates following ASP code:

```
def_use_pair(S1, S2, O)
    :- def(S1, O), use(S2, O),
    reaches(S1, S2), not redefs(S1, _, S2, O).
redefs(S1, S2, S3, O)
    :- S1 != S2, S2 != S3,
```

```
    def(S1, O),  def(S2, O), use(S3, O),
    reaches(S1, S2), reaches(S2, S3).
```

multi_path_def_clear_path_def

Generates ASP code to determine *def-clear-paths* on all possible paths. This directive generates the following ASP rule:

```
def_clear_path(S1, S2, O)
    :- object(O), edge(S1, S2), not def(S2, O).
def_clear_path(S1, S2, O)
    :- object(O), edge(S1, S3), not def(S3, O),
    def_clear_path(S3, S2, O).
def_use_pair(S1, S2, O)
    :- object(O), def(S1, O), use(S2, O), edge(S1, S2).
def_use_pair(S1, S2, O)
    :- object(O), def(S1, O), use(S2, O), edge(S3, S2),
    def_clear_path(S1, S3, O).
```

any_order_t1_tuple_t2 ${‘t1’, ‘t2’}$

Generates ASP code to determine all testable entities to be covered under any ordered $(t1, t2)$-way path aware coverage criterion. For example, this directive generates following ASP code,

when $t1 = 2$ and $t2 = 2$,

```
reaches(S, S) :- state(S).
any_order_2_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    any_order(S1, S2).
any_order_2_tuple_2(S, O1, V1, S, O2, V2)
    :- O1 < O2,
    option(S, O1, V1), option(S, O2, V2).
any_order(A, B) :- reaches(A, B).
```

when $t1 = 2$ and $t2 = 3$,

```
reaches(S, S) :- state(S).
any_order_2_tuple_3(S, O1, V1, S, O2, V2, S, O3, V3)
    :- O1 < O2, O2 < O3,
    option(S, O1, V1), option(S, O2, V2), option(S, O3, V3).
any_order_2_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
    :- S1!=S3, O1 < O2,
    any_order(S1, S3),
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3).
any_order_2_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1!=S2, O2 < O3,
    any_order(S1, S2),
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3).
any_order(A, B, C) :- reaches(A, B), reaches(B, C).
```

when $t1 = 3$ and $t2 = 2$,

```
reaches(S, S) :- state(S).
any_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2), any_order(S1, S2, _).
any_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    any_order(S1, _, S2).
any_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    any_order(_, S1, S2).
any_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 == S2, O1 < O2,
    option(S1, O1, V1), option(S2, O2, V2).
any_order(A, B, C) :- reaches(A, B), reaches(B, C).
```

when $t1 = 3$ and $t3 = 2$,

```
reaches(S, S) :- state(S).
any_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S3, O3, V3)
    :- S1 != S2,  S2 != S3,
    option(S1, O1, V1), option(S2, O2, V2), option(S3, O3, V3),
```

```
    any_order(S1, S2, S3).
any_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S3, O3, V3)
    :- S1 == S2, S2 != S3, O1 < O2,
    option(S1, O1, V1), option(S2, O2, V2), option(S3, O3, V3),
    any_order(S1, S2, S3).
any_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S3, O3, V3)
    :-  S1 != S2, S2 == S3, O2 < O3,
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3),
    any_order(S1, S2, S3).
any_order_3_tuple_3(S, O1, V1, S, O2, V2, S, O3, V3)
    :-  O1 < O2, O2 < O3,
    option(S, O1, V1), option(S, O2, V2), option(S, O3, V3).
any_order(A, B, C) :- reaches(A, B), reaches(B, C).
```

consecutive_order_t1_tuple_t2 ${‘t1’, ‘t2’}$

Generates ASP code to determine all testable entities to be covered under consecutive ordered $(t1,t2)$-way path aware coverage criterion. For example, this directive generates following ASP code,

when $t1 = 2$ and $t2 = 2$,

```
edge(S,S) :- state(S).
consecutive_order_2_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    consecutive_order(S1, S2).
consecutive_order_2_tuple_2(S, O1, V1, S, O2, V2)
    :- O1 < O2,
    option(S, O1, V1), option(S, O2, V2).
consecutive_order(A, B) :- edge(A, B), A != B.
```

when $t1 = 2$ and $t2 = 3$,

```
edge(S,S) :- state(S).
consecutive_order_2_tuple_3(S, O1, V1, S, O2, V2, S, O3, V3)
    :- O1 < O2, O2 < O3,
    option(S, O1, V1), option(S, O2, V2), option(S, O3, V3).
consecutive_order_2_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
```

```
    :- S1 != S3, O1 < O2,
    consecutive_order(S1, S3),
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3).
consecutive_order_2_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1!=S2, O2 < O3,
    consecutive_order(S1, S2),
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3).
consecutive_order(A, B, C) :- edge(A, B), edge(B, C), A != B, B != C.
```

when $t1 = 3$ and $t2 = 2$,

```
edge(S,S) :- state(S).
consecutive_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    consecutive_order(S1, S2, _).
consecutive_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    consecutive_order(_, S1, S2).
consecutive_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    consecutive_order(S1, _, S2).
consecutive_order_3_tuple_2(S, O1, V1, S, O2, V2)
    :-  O1 < O2,
    option(S, O1, V1), option(S, O2, V2).
consecutive_order(A, B, C) :- edge(A, B), edge(B, C), A != B, B != C.
```

when $t1 = 3$ and $t2 = 3$,

```
edge(S,S) :- state(S).
consecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S3, O3, V3)
    :- S1 != S2,  S2 != S3,
    option(S1, O1, V1), option(S2, O2, V2), option(S3, O3, V3),
    consecutive_order(S1, S2, S3).
consecutive_order_3_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
    :- S1 != S3, O1 < O2,
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3),
    consecutive_order(S1, _, S3).
consecutive_order_3_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
```

```
    :- S1 != S3, O1 < O2,
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3),
    consecutive_order(S1, S3, _).
consecutive_order_3_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
    :- S1 != S3, O1 < O2,
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3),
    consecutive_order(_, S1, S3).
consecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1 != S2, O2 < O3,
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3),
    consecutive_order(S1, S2, _).
consecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1 != S2, O2 < O3,
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3),
    consecutive_order(S1, _, S2).
consecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1 != S2, O2 < O3,
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3),
    consecutive_order(_, S1, S2).
consecutive_order_3_tuple_3(S, O1, V1, S, O2, V2, S, O3, V3)
    :-  O1 < O2, O2 < O3,
    option(S, O1, V1), option(S, O2, V2), option(S, O3, V3).
consecutive_order(A, B, C) :- edge(A, B), edge(B, C), A != B, B != C.
```

nonconsecutive_order_t1_tuple_t2 ${'t1', 't2'}$

Generates ASP code to determine all testable entities to be covered under non-consecutive ordered $(t1,t2)$-way path aware coverage criterion. For example, this directive generates following ASP code,

when $t1 = 2$ and $t2 = 2$,

```
edge(S,S) :- state(S).
nonconsecutive_order(A, B)
    :- reaches(A, C), reaches(C, B),
    A != B, B != C.
nonconsecutive_order_2_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
```

```
    option(S1, O1, V1), option(S2, O2, V2),
    nonconsecutive_order(S1, S2).
nonconsecutive_order_2_tuple_2(S, O1, V1, S, O2, V2)
    :- O1 < O2,
    option(S, O1, V1), option(S, O2, V2).
```

when $t1 = 2$ and $t2 = 3$,

```
edge(S,S) :- state(S).
nonconsecutive_order_2_tuple_3(S, O1, V1, S, O2, V2, S, O3, V3)
    :- O1 < O2, O2 < O3,
    option(S, O1, V1), option(S, O2, V2), option(S, O3, V3).
nonconsecutive_order_2_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
    :- S1!=S3, O1 < O2,
    nonconsecutive_order(S1, S3),
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3).
nonconsecutive_order_2_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1!=S2, O2 < O3,
    nonconsecutive_order(S1, S2),
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3).
nonconsecutive_order(A, B, C)
    :- reaches(A, B), reaches(B, X), reaches(X, C),
    A != B, B != C, C != X.
nonconsecutive_order(A, B, C)
    :- reaches(A, X), reaches(X, B), reaches(B, C),
    A != B, B != C, B != X.
```

when $t1 = 3$ and $t2 = 2$,

```
edge(S,S) :- state(S).
nonconsecutive_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    nonconsecutive_order(S1, S2, _).
nonconsecutive_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
    nonconsecutive_order(S1, _, S2).
nonconsecutive_order_3_tuple_2(S1, O1, V1, S2, O2, V2)
    :- S1 != S2,
    option(S1, O1, V1), option(S2, O2, V2),
```

```
    nonconsecutive_order(_, S1, S2).
nonconsecutive_order_3_tuple_2(S, O1, V1, S, O2, V2)
    :- O1 < O2,
    option(S, O1, V1), option(S, O2, V2).
nonconsecutive_order(A, B, C)
    :- reaches(A, B), reaches(B, X), reaches(X, C),
    A != B, B != C, C != X.
nonconsecutive_order(A, B, C)
    :- reaches(A, X), reaches(X, B), reaches(B, C),
    A != B, B != C, B != X.
```

when $t1 = 3$ and $t2 = 3$,

```
edge(S,S) :- state(S).
nonconsecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S3, O3, V3)
    :- S1 != S2,  S2 != S3,
    option(S1, O1, V1), option(S2, O2, V2), option(S3, O3, V3),
    nonconsecutive_order(S1, S2, S3).
nonconsecutive_order_3_tuple_3(S, O1, V1, S, O2, V2, S, O3, V3)
    :-  O1 < O2, O2 < O3,
    option(S, O1, V1), option(S, O2, V2), option(S, O3, V3).
nonconsecutive_order_3_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
    :- S1 != S3, O1 < O2,
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3),
    nonconsecutive_order(S1 ,_, S3).
nonconsecutive_order_3_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
    :- S1 != S3, O1 < O2,
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3),
    nonconsecutive_order(S1, S3, _).
nonconsecutive_order_3_tuple_3(S1, O1, V1, S1, O2, V2, S3, O3, V3)
    :- S1 != S3, O1 < O2,
    option(S1, O1, V1), option(S1, O2, V2), option(S3, O3, V3),
    nonconsecutive_order(_, S1, S3).
nonconsecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1 != S2, O2 < O3,
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3),
    non-consecutive_order(S1, _, S2).
nonconsecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1 != S2, O2 < O3,
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3),
```

```
    nonconsecutive_order(S1, S2, _).
nonconsecutive_order_3_tuple_3(S1, O1, V1, S2, O2, V2, S2, O3, V3)
    :- S1 != S2, O2 < O3,
    option(S1, O1, V1), option(S2, O2, V2), option(S2, O3, V3),
    non-consecutive_order(_,S1, S2).
nonconsecutive_order(A, B, C)
    :- reaches(A, B), reaches(B, X), reaches(X, C),
    A != B, B != C, C != X.
nonconsecutive_order(A, B, C)
    :- reaches(A, X), reaches(X, B), reaches(B, C),
    A != B, B != C, B != X.
```